Forth in der Robotik

Manuel Rodriguez

9.Oktober 2017

Unpublished manuscript

Zusammenfassung

Forth hat keine Standard-Bibliotheken und die Hardware ist nicht compiler-friendly. Das macht Forth zu einer Toy-Language. Man kann damit keine größeren Projekte realisieren und für die Robotik ist Forth ungeeignet. Will man in Forth programmieren, fallen extrem hohe Kosten an.

Stichworte: Forth, Robotics

Inhaltsverzeichnis

1 Einleitung

1.1 Wozu Forth?

Anfangs war ich etwas skeptisch was Forth angeht. Als ich die Sprache und die dazugehörigen Anleitungen das erste Mal gesehen habe, wusste ich sofort: das ist nichts für mich. Jeder, der Forth das

erste Mal sieht muss es hassen. Es ist einfach zu sondernbar, zu umständlich in der Benutzung als das man sich freiwillig näher damit auseinandersetzt. Doch irgendwie ist es nicht dabei geblieben, sondern irgendwann wollte ich dann doch wissen wie das genau geht.

Aber was genau ist Forth eigentlich? Am ehesten kann man es für Neueinsteiger so erklären, dass man es anhand von klassischen Programmiersprachen, klassischen CPUs und klassischen Compilern angeht. Wenn man ein Programm in C schreibt, jagd man es anschließend durch den Compiler, der macht daraus Maschinencode und dieser wird auf der CPU ausgeführt. C gilt als sehr effiziente Sprache, der damit erzeugte Machinencode ist nahe dran am Optimium. Und hier kommt Forth ins Spiel. Forth liegt genau zwischen C und Maschinenopcodes. Es ist eine Art von Makroassembler, nur um einiges mächtiger. Was das heißt kann man am besten erklären wenn man ein C Programm auf einem Commodore 64 zum laufen bringen möchte. Das ist deshalb so interessant weil auf dem C-64 eben kein Linux läuft und auch kein GCC Compiler. Das übliche Vorgehen dafür geht so, dass man auf einem normalen PC das Programm mittels Crosscompiler in Maschinencode übersetzt und diese Assembler-Datei dann auf dem C-64 ausführt. Man braucht also neben dem C-64 noch einen vollwertigen Computer wo der eigentliche Compiler drauf ist.

Forth hingegen bringt seinen eigenen Compiler mit. Es ist so eine Art von Entwicklungsumgebung die eine Bash und eine Programmiersprache beinhaltet und auf jedem Kleincomputer ausgeführt werden kann. Viele C-64 User werden mit Assemblern schon Bekanntschaft gemacht haben, neben BASIC war das die wichtigste Sprache und üblicherweise wurden Assembler auf dem C-64 direkt ausgeführt. Der Nachteil wenn man direkt in Assembler programmiert besteht darin, dass man das komplette Programm darin schreiben muss. Man fängt in Zeile 1 mit Mnemonic wie LDA zu arbeiten und die letzte Zeile ist ebenfalls nach diesem Raster aufgebaut. Das ist zwar möglich, aber umständlich. Bei Forth hingegen arbeitet man nach dem Bootstrapping Verfahren. Damit ist gemeint, dass man zuerst einmal das Forth an den C-64 anpasst, also ein Mapping herstellt zwischen den Forth Standard-Befehlen und den MOS 6502 Opcode. Jetzt braucht man schon nicht mehr LDA #irgendwas hinzuschreiben sondern kann in einer Forth Hochsprache programmieren. Aber, von Hause aus ist Forth nicht besonders mächtig. Beispielsweise fehlt eine Funktion um Arrays zu verwalten. Aber auch das ist kein Problem, weil man das Forth um diese Funktion nachrüstet. Wie das geht erläutert [1] Dort ist erklärt wie man Arrays anlegt und daraus einen neuen Befehl macht. Die Idee ist es, das Forth zu erweitern, bis man am Ende unter Forth ähnlich komfortabel programmieren kann wie in Python. Wohlgemerkt, ohne dass man dafür dezidierte C-Compiler oder Betriebssysteme benötigt. Sondern ein simples Standard-Forth plus selbstprogrammierte Erweiterungen sind alles was man benötigt.

Forth zeichnet sich durch einen extremen Minimalismus aus. Am ehesten kann man die Sprache mit Brainfuck vergleichen, nur mit dem Unterschied das Forth erweiterbar ist. In seiner Grundfunktion kann Forth beispielsweise keine Objekte verwalten. Was Forth jedoch kann ist auf den RAM zugreifen. Also einzelne Werte auslesen und Speichern. Daraus kann man sich selber seine eigene OOP Erweiterung bauen. Das klingt jetzt erstmal aufwendig, aber man braucht dafür erstaunlich wenig Lines of Code. Und das macht die Faszination des ganzen aus. Anders als bei Linux gibt es keine POSIX Standards, es gibt auch keine C99 Standards, sondern man muss sich das Forth so komfortabel oder so puristisch einrichten wie man es benötigt.

Objektiv betrachtet macht Forth keinen Sinn, weil man mit Python sehr viel schnellere Ergebnisse erzielt. Wer einfach nur ein kleines Programm schreiben will, der ist mit Python besser aufgehoben. Forth jedoch bietet Vorteile wenn man sich für Computer interessiert, und grundsätzlich etwas darüber lernen möchte. Bei Python und den meisten anderen Hochsprachen ist die eigentliche Funktion sehr gut versteckt. Den Python Interpreter hat jemand geschrieben, und der Python Interpreter wiederum wurde mit einem C-Compiler nach Assembler konvertiert, den wiederum ein anderer geschrieben usw. usw. Bei Forth hingegen ist man direkt auf der Hardware, und wem das nicht Lowlevel genug ist, kann sich in VHDL seinen eigenen FPGA Prozessor bauen.

Die Linux OpenSource Bewegung ist bereits relativ Hacking-orientiert eingestellt, indem Sinne dass es eine kutlurelle Übereinkunft gibt, keine propritäre Software von Microsoft oder Borland einzusetzen und man stattdessen seinen eigenen Kernel und seinen eigenen Compiler verwendet. Aber, Linux kopiert nur das vorhandene Ökosystem, es macht das selbe was auch Unix und Windows tun, nur eben mit freiem Sourcecode. Forth geht einen Schritt weiter. Forth bezieht die Hardware mit ein und verfolgt einen konsquenten Minimalismus. Das heißt, es gibt überhaupt keine Geheimnisse mehr, Forther wissen alles. Oder zumindest ist das der Anspruch den die Community verfolgt.

Für die ersten Gehversuche würde ich von den meisten Forth Implementierungen wie gforth, Jonesforth oder Fig-forth abraten und stattdessen das Online-Portal Repl.it [2] empfehlen. Man erhält dort einen Forth Interpreter in Javascript, braucht bei sich nichts zu installieren und kann die meisten Beispiele damit ausführen. Begleitend dazu empfehle ich eine kleine Textdatei anzulegen “Forthhandbuch.txt”, in die man wichtige Befehle die man gerade gelernt notiert. Anders als bei normalen Programmiersprachen und normalen Betriebssystemen kann man bei Forth nichts herunterladen und das auf seiner Festplatte installieren, sondern im wesentlichen ist Forth ein Bildungsinhalt, also was man lernen und anwenden muss. Die Idee ist folgende: nach einem Atomangriff der einen EMP Blitz beinhaltete sind sämtliche Festplatten gelöscht und auch alle Kopien des Linux Betriebssystems vernichtet. Propritäre Microsoft-Software gibt es auch keine. Jetzt sind die Forth Leute in ihrem Element. Sie holen ihre Notizen aus dem Schrank, tippen die Forth Kommandos auf eigens konstruierten PCs ein und booten damit nicht einfach die Maschine sondern beim Booten schreiben sie on-the-fly auch gleich die komplette Software. Sie entfalten damit die komplette Betriebssystem-Software, alles was man braucht um Spiele zu spielen, Datenkommunikation zu betreiben oder Berechnungen durchzuführen. Das geht mit Linux nicht, dort braucht man zumindest den Sourcecode der bei einem aktuellen Fedora rund 10 Gigabyte beträgt. Sowas kann man schlecht ausdrucken.

1.2 Das erste Forth Programm

Anstatt dem üblichen Forth Style zu pflegen und komplett unverständlichen Code zu schreiben, möchte ich als Einführungsbeispiel ein kleines Programm vorstellen, was gut lesbar ist. Es funktioniert nach dem gebräuchlichen Eingabe->Verarbeitung->Ausgabe Scheme und ist in Forth ausführbar. Leider ist Lyx nicht in der Lage Forth-Syntax korrekt zu formatieren, so dass es als Nur-Text erscheint.

Auf den ersten Blick ist das ganze nichts besonderes, der Aufbau erinnert an ein C-Programm bei dem irgendwas fehlt. Dieses etwas ist die Infixnotation, das man also normalerweise schreibt:

```
printf(%d,number)
```

Diese Möglichkeit gibt es bei Forth leider nicht. Stattdessen läuft alles über den imaginären Stack, den man sich als eine Art von Candyautomat vorstellen kann, bei dem man oben etwas drauflegt und dort auch wieder herunternehmen muss. Dennoch hat Forth gegenüber C einen gewaltigen Vorteil: es läuft überall. Den obigen

[1] https://learnxinyminutes.com/docs/forth/

[2] https://repl.it/languages/forth

```
variable number

: input
  ." Please enter a number: "
  23 number !
  number @ . cr \ get number and print

;
: calculation
  number @  \ get number
  100 +  \ add 100
  number ! \ store number to memory
  ." I add 100 to your number" cr \ tell what i'm
      doing
;
: output
  ." Output="
  number @ \ get number
  . cr \ print to console
  ." thanks for using Forth"
;

: main
  input
  calculation
  output
;

main
```

Abbildung 1: Hello World in Forth

Sourcecode kann man sowohl unter Linux, Windows und sogar auf einem C-64 ausführen. Er braucht nicht in Machinenopcodes übersetzt werden, und man braucht auch keine Python-Interpreter oder virtuelle Maschinen. Das ist bei Forth alles schon eingebaut. Das klingt zunächst etwas ungewohnt, aber es ist die Zukunft.

Doch gehen wir das Programm im Detail durch. Was es tut ist selbsterklärend, man kann es aus dem Sourcecode ablesen. Das eigentlich interessante ist das, was es nicht macht. Genauer gesagt wirkt die fehlende Infix Notation auf den Einsteiger irrtierend. Er sieht das Programm, aber er kann damit nichts anfangen. Weil, die Mainfunktion übergibt keine Parameter. Nun gut, auch in C gibt es Funktionen die ohne Parameter aufgerufen werden, weil sie entweder globale Variablen verwenden oder wie in C++ auf Methoden der selben Klasse zugreifen. Nur, es gibt in Forth auch sonst keine Funktionen die Parameter verwenden. Auch das Word, was etwas auf den Bildschirm ausgibt kommt ohne Parameter aus. Stattdessen läuft im Hintergrund etwas anderes mit: der Stack.

1.3 Der Forth Interpreter

Bisher wurde unterstellt, dass eine lauffähige Forth Virtual Maschine bereits vorliegt. Wenn man sich mit Forth näher beschäftigen will, ist es erforderlich, diese in C selber zu programmieren. Oder zumindest fertigen Code von github zu nehmen und ihn mittels “gcc forth.c” zu kompilieren. Hier [3] gibt es eine Forth Virtual Maschine zum Download sie umfasst nur 27 kb. Wenn man sie kompiliert erhält man eine 36 kb große “a.out” Datei. Was sie macht kann man ungefähr anhand des Sourcecodes ahnen. Man kann dort eintippen “1 2 + .” und sieht dann das Ergebnis. Ferner kann man eigene Words definieren und Schleifen ausführen. Von der Performance her würde ich sagen, dass das Forth ca. 20x langsamer ist als gforth. Spannender ist jedoch eine andere Frage, und zwar ist das obige Forth eine Virtual Maschine welche in C programmiert wurde. Also ein kompletter Computer, vergleichbar mit qemu, na ja fast jedenfalls. Am ehesten kann man die Forth Implementierung mit einem Turing-Machine-Simulator vergleichen. Wo also die mögilchen Befehle auf ein Minimum reduziert sind, so dass man gerade so Programmieren damit kann.

Ich erwähne eine Forth Virtual Machine deshalb um etwas über die Geschichte von Forth zu erzählen. Es wurde Anfang der 1970’er Jahre von Charles H Moore und Elizabeth Rather erfunden, zu einer Zeit als längst die ersten Computer gab. Die Forth Virtual Maschine war der Versuch in die Vergangenheit zu blicken, also in Software das nachzuahmen, was Ende der 1930’er erstmals gebaut wurden, Turiing-mächtige Computer. Man kann Forth als weiterentwickelte Turing-Maschine verstehen. Eine Turing-Maschine ist ein didaktisches Modell um die Funktionsweise von Computern zu erläutern. Es wird meist ein Band plus Schreiblese Kopf verwendet. Damit kann man die Grundlagen der Informatik erläutern. Aber, das didaktische Konzept Turing-Maschine hat den Nachteil, dass es kein richtiger Prozessor ist und auf dem Band gibt es auch keine Bereiche wo man Unterprogramme ablegen kann. Forth ist ein wenig detailierter und orientiert sich stärker an einem richtigen Computer.

Theoretisch, und das ist eine gute Übung für den Neueinsteiger, kann man mit dem oben verlinkten Forth eigene kleine Programme schreiben. Anders als bei einem reinem CPU Simulator also einen Algoirthmus eingeben und den ausführen. Forth dient also dazu, etwas über Computer zu lernen. Und wenn man nicht nur die Forth Virtual Maschine in C programmiert sondern sie als FPGA implementiert, lernt man zusätzlich noch etwas über Hardware.

Aber, genau hier zeigt sich auch was die Schwäche von Forth ist. Es wurde Anfang der 1970’er erfunden bzw. vorgestellt zu einer Zeit als es bereits Hochsprachen gab. Die Mother of all Demos wurde zwei Jahre früher abgehalten (1968), es gab also bereits Computer mit einer GUI. Forth war also keine weiterentwicklung oder für den Produktiveinsatz gedacht, um damit Software zu entwickeln, sondern Forth war das selbe, was wir heute als Rasberry PI kennen, eine Lernumgebung damit sich Laien, Interessierte oder wer auch immer spielerisch mit der Informatik auseinandersetzen kann.

Und jetzt der Sprung zurück in die Gegenwart. Warum fasziniert Forth bis heute, wenn es doch schon so alt ist? Der Grund ist, dass Forth eine sehr gute Lernumgebung ist. Wenn man etwas über Compiler und Computerhardware lernen möchte sogar die weltweit beste. Der Clou ist nicht so sehr das oben verlinkte C-Programm, es ist eher banal, und der Clou liegt auch nicht in den zahlreichen anderen Forth Implementierungen, sondern das bemerkenswerte entsteht, wenn man mit Leuten redet, die sich intensiv mit Forth beschäftigt haben. Üblicherweise ist das gleichbedeutend damit, dass sie Computertechnik grundlegend verstanden haben und in Folge dessen so wunderbare Chips entwickeln können wie den GA144. Forth ist also primär ein Lerninstrument wodurch man Wissen erlangt was genutzt wird um High-End-Hardware zu entwickeln.

Es geht also nicht wirklich darum, Forth zu verstehen, sondern alles andere was mit Informatik zu tun hat. Also Software, Compiler, Betriebssysteme, Transistoren usw. Und auf dieser Basis erklärt sich auch, wie die Forth Community funktioniert. Es geht weniger darum, mit Forth etwas sinnvolles anzufangen, sondern die Veröffentlichungen und Projekte orientieren sich an dem Lernbedarf. Das heißt, wenn jemand wissen möchte, wie man einen Parallelcomputer baut, dann wird er das anhand von Forth untersuchen. Er könnte zwar auch eine normale Turing-Maschine nehmen und sich vorstellen, dass es da ein Band gibt und viele Leseköpfe, aber das Abstraktionsniveau von Forth ist besser. Weil man dort adhoc einen kompletten Primzahlalgorithmus implementieren kann. So kann man testen ob das eigene Design funktioniert, was man sich überlegt hat.

[3] https://gist.github.com/lbruder/10007431

1.4 Ein Hoch auf Lisp und Forth

Noch in den 1980'er gehörten LISP und Forth zu den beiden wichtigsten Sprachen. Man setzte sie auf hocheffizienten Stackmachinen ein und führte Projekte im Bereich der Künstlichen Intelligenz durch. Wichtige und große Projekte waren das, von der die normale Welt keine Ahnung hatte. Es gab damals sogar humanoide Roboter die in LISP programmiert wurde und zwar auf Workstations die über Bootstrapping zuerst ein Forth starteten und dann ein Common-Lisp Betriebssystem. Das waren nicht so Anfänger Maschinen wie sie heute verbreitet sind, sondern es war advanced Technologie. Wir reden hier nicht über VLSI oder 7 nm CPUs wie sie aktuell von Samsung gefertigt werden, nein damals gab es optische Computer, also die nächste Generation an Schaltkreisen welche das Silizium verdrängt haben. Die Software in den 1980'er lief ultrastrabil, damit ist gemeint, dass das Problem mit den Bugs schon im Jahrzehnt davor gelöst wurde und ausschließlich Metaprogramming eingesetzt wurde. Das führte zu sehr zuverlässiger Software die Modellgetrieben entwickelt wurde. Nicht so wie die heutige Software, wo Menschen dran rumprogrammieren sondern damals wurde die Software aus First Order Logik-Spezifikationen synthesiert und die wiederum wurden durch leistungsfähige Neuronale Netz Supercomputer laufend optimiert.

Ja so war es damals in den 1980'er Jahren. Damals waren die Straßen sauber, das Essen schmeckte und der technische Fortschritt war überall. Es war ein sehr schönes erfolgreiches Jahrzehnt, in dem die Maschinelle Intelligenz zu ihrer Blüte geführt wurde.Man kann sich das so vorstellen, dass damals die Ingineure rund 2 Monate brauchten und dann lief das Robot-Control-System in Forth perfekt. Es war unglaublich schnell, es war fehlerfrei und der Roboter war in der Lage menschliche Sprache zu verstehen. Er konnte sie buchstäblich von den Lippen ablesen und war von einem Menschen nicht zu unterscheiden. Einige der Roboter wurden sogar mit ins Entwicklungsteam eingebunden und verrichteten dort die selben Aufgaben die auch Menschen ausführten. Und es funktionierte hervorragend. Niemals gab es Probleme mit den Androiden, sie kamen morgens pünktlich zur Besprechung, arbetieten dann konzentriert bis zum Mittagessen durch und waren in den Pausen immer zu einem Smalltalk bereit. So wie es in einer idealen Gesellschaft üblich ist wurden Probleme schon vor ihrem Entstehen ausgeräumt und Einsicht in die Abläufe bei allen Angestellten gefördert. Der einzelne setzte sich für sein Team ein und das Team profitierte davon. Maschinelle Intelligenz und humanoide Intelligenz gingen Hand in Hand und verstärkten sich gegenseitig. Es war ein angenehmes Arbeiten mit den allerneuesten Technologien. Dazu gehörten fortschrittliche Betriebssysteme, neuartige Animationssoftware, Expertensysteme sowie CAD Programme um die nächste Generation an Hardware zu entwickeln. Auch an bionische Implantante auf Basis des Forth Stackprozessors war gedacht. Die Verbindung mit dem Hive-Bewusstsein funktionierte reibungslos.

1.5 Literaturangaben

Die Literatur zu Forth ist umfangreich. Als Besonderheit muss man sagen, dass anders als bei vielen anderen Programmiersprachen der Großteil der Literatur aus den 1980'er Jahren stammt. Auf den Einsteiger wirkt das so, als wenn Forth überholt wäre. Nur, die Paper aus dieser Zeit sind bis heute aktuell, ja mehr noch, je mehr Zeit vergeht desto interessanter werden sie. Hier eine kurze Auflistung was man gelesen haben sollte, um mitreden zu können:

- Referenzwerk über die Vorteile von Stackcomputern / aka Forthchips, “Stack computers: the new wave" [8], online auf https://users.ece.cmu.edu/∼koopman/stack_computers/index.html
- Noch ein Werk über die Geschichte der Stackcomputer, [9]
- Videoaufzeichnungen von Forth Vorträgen auf deutsch natürlich, https://wiki.forth-ev.de/doku.php/events:start
- Videoaufzeichnung der Silicon Valley Forth User group, https://www.youtube.com/channel/UC2v6b9814uIA5egk5-yHAVw/videos
- Inhaltsverzeichnis der deutschen Forth Zeitschrift, https://wiki.forth-ev.de/doku.php/projects:4dinhalt
- Online Lexikon mit dem Schwerpunkt Forth und alternative Programmierkonzepte, http://wiki.c2.com/
- Online Portal Repl.it zum Ausführen von Forth Code, https://repl.it/languages/forth
- The Journal of Forth Application and Research, http://soton.mpeforth.com/flag/jfar/index.html

Eine Besonderheit von Forth ist es, dass es keine Gegenstimmen gibt. Man wird also keine Literatur finden welche sich kritisch mit Forth auseinandersetzt. Und wenn doch, hat der Autor das Konzept nicht verstanden. Alle, die sich sachlich inhaltlich mit Forth auseinandersetzen tun dies mit einer Pro-Forth Haltung.

2 Programmieren

2.1 Geschwindigkeit

Kommen wir jetzt zu einem Thema was für Neueinsteiger besonders aufregend ist, und zwar die Frage der Performance. Die schlechte Nachricht vorneweg. Forth ist nicht schneller als C++. In der Mehrzahl der Fälle ist Forth gegenüber C/C++ sogar langsamer. Die Ursache dafür ist, dass ein Forth system nur die selben CPU-Opcodes aufrufen kann die auch ein per C++ Compiler erzeugtes Programm aufruft. Und da heutige C/C++ Compiler sehr effizienten Maschinencode erstellen kann man mit Forth das nicht großartig verbessern.

Aber, wenn man in Forth programmiert fallen einem sehr viel mehr Möglichkeiten ein, wie man die Hardware und die Software schneller macht. Hat man erstmal seinen Primzahlen Algorithmus als Forth-Code vorliegen, grübelt man automatisch darüber nach, wie denn die dazu passende CPU aussehen mag. Und relativ schnell bemerkt man, dass der Flaschenhals nicht so sehr die Software ist, sondern dass der Flaschenhals die x86 CPU ist. Verwendet man eine Forth kompatible CPU wird das selbe Programm sehr viel schneller ausgeführt.

Und wenn man das geschafft hat ist der nächste Schritt die Forth CPU mit anderen Forth CPU als Array zu verbinden und eine clockless Architektur zu verwenden, wodurch sich weitere Performance-Verbesserungen ergeben. Anders ausgedrückt, Forth ist zwar noch nicht per se schneller, aber innerhalb des Forth Universum ist man eher in der Lage daran etwas zu ändern.

Die Linux und Windows User sind hingegen in ihr System eingesperrt. Was der C-Compiler mit ihrem Code macht, darauf haben sie keinen Einfluss, und wenn Intel beim nächsten Update keine CPUs veröffentlicht die dramatisch schneller sind können sie auch nichts machen. So entsteht ein Fatalismus, wo man glaubt als normaler Programmierer ohnehin nichts verändern zu können. Man richtet sich stattdessen ein und glaubt es müsste so sein, dass ein normaler Desktop PC stolze 20 Watt schluckt obwohl er gar nichts rechnet sondern nur auf den Userinput wartet.

Durexforth Wie Forth intern funktioniert kann man am besten anhand von Durexforth [4] erkennen. Durexforth ist zunächst einmal eine

[4] https://github.com/jkotlinski/durexforth

.ASM Datei die auf dem C-64 assembliert wird. In dieser ASM Datei gibt es verschiedene Unterprogramme. Eines für Fetch, eines für Store und für jedes andere Forth Statement ebenfalls. Das jeweilige Unterprogramm ist in Assembler geschrieben. Meist enthält es 4-10 Assemblerbefehle. Wenn man Durexforth startet und dort dann ein Forth Programm startet werden diese Assemblerbefehle ausgeführt. Das heißt, die Performance von Durexforth ist vergleichbar mit BASIC, es ist eine langsame Hochsprache die zur Laufzeit interpretiert wird. Vermutlich ist die Performance von Durextforth um den Faktor 2-3 langsamer als würde man mit CC65 ein Programm für den C-64 schreiben. Die Performance hängt letzt davon ab, welche Maschinenbefehle nacheinander aufgerufen werden. Genau genommen ist Forth also eine Virtuelle Maschine, vergleichbar mit der JVM. Es ist ein Konverter der Forth Code nimmt und ihn für eine konkrete CPU aufbereitet, daher auch die Verzögerung.

Ähnlich wie die fehlende Infix Notation ist die fehlende Performance von Forth etwas, was irritierend auf Einsteiger wirkt. Dadurch wird etwas deutlich, und zwar dass es einen Bias gibt. Die meisten Programmierer haben sich daran gewöhnt, dass die CPU einfach da ist und funktioniert. Sie sind froh darüber, dass Intel oder ein anderer Hersteller sie liefert und das der Instruction Set vorgegeben ist. Üblicherweise wollen Java Programmierer in Java programmieren und nicht in Opcodes einer AMD CPU. Deshalb hat man sich auf Standards geeignet. Der MOS 6502 hat klar definierte Opcodes und auch die AMD64 Architektur hat feste Opcodes. Forth hält sich nicht an diese Regeln, Forth macht etwas eigenes.

Anders formuliert, durch die fehlende Infix Notation ist Forth schwerer verständlich, und durch die fehlende Performance sind Forth Programme langsamer. Trotzdem ist das aber besser. Weil man als User die Kontrolle über sein System zurückbekommt. Man ist nicht länger abhängig von Linux und man ist nicht abhängig von Intel.

2.2 Objekt-orientierte Programmierung

Wie man anhand des Hello World Programs gesehen hat, ist es komfortabel möglich mit Forth ausführbare Programme zu schreiben. Man muss sich zwar etwas umgewöhnen aber es funktioniert irgendwie. Gänzlich anders sieht es bei der Objektorientierten Programmierung aus. Seit Ende der 1980'er sind eine Reihe von Papern erschienen in denen OOP für Forth diskutiert wurde. Manchmal mit abgedruckten Sourcecode manchmal ohne. Die Schwierigkeit besteht darin, dass OOP relativ aufwendig ist zu implementieren. Das ganze ist nicht nur ein simples Array was man im Speicher reservert, sondern man benötigt auch Vererbung, Garbage Collection und Überladen. Bis heute gibt es keine brauchbare Extension für Forth was das Thema zufriedenstellend behandelt. Das heißt nicht, dass Forth und OOP nicht zusammenpassen ganz im Gegenteil. Vielmehr ist hier noch Raum für Leute die zeigen wollen was sie draufhaben. Eine OOP Erweiterung für Forth zu programmieren gilt als anspruchsvoll wie ergiebig gleichermaßen.

Einsatzfähig aktuell sinid gerade nochso Structs. Es gibt mehrere Beispiele wie man das in Forth realisiert. Man muss im Grunde eine kleine Forth Extension Programmieren und hat dann in Forth einen Struct vergleichbar wie in C oder in Python. Aber echte OOP ist das nocht nicht. Wer nicht gerade selbst die benötigte Extension programmieren möchte kann aktuell unter Forth leider keine Objektorientierte Programmierung verwenden. Er kann zwar manuell viele Words definieren aber das ist dann nur so als wenn man in C programmiert. Man schreibt also 100 Unterfunktionen, kann die auch aufrufen aber es fällt schwer damit größere Programme zu schreiben. Man darf nicht vergessen, dass sich OOP nicht ohne Grund durchgesetzt hat. Es erleichtert die Erstellung von längeren Programmen ungemein.

Als Nachteil von Forth würde ich die fehlende OOP-Funktionalität dennoch nicht auslegen. Sondern eher als unerledigte Aufgabe, dass jemand der wirklich viel Freizeit hat, sich des Problems annimmt und so endlich auch in Forth das realisiert was in Java, Python und C# längst Standard ist. Der Grund warum sich bisher noch keiner an diese Aufgabe gewagt hat liegt daran, dass OOP zurecht als anspruchsvoll gilt. Auch der C++ Compiler der in Linux verwendet wird gilt als anspruchsvoll. Es macht eben einen Unterschied ob man nur simples C in Maschinencode übersetzt oder ob man das selbe für C++ durchführt. Auch einen Java Compiler bzw. Interpreter ist keine leichte Aufgabe, und wenn man das in Forth machen will wird es richtig anspruchsvoll. Bislang gibt es nur mehrere gescheiterte Versuche OOP in Forth zu realisieren. Eine davon ist mini-oop.fs welche vorgibt eine objektorientierte Erweiterung zu sein, defakto jedoch keine Mehrfachvererbung und keine Stringzuweisung unterstützt.

Jetzt gibt es zwei Möglichkeiten damit umzugehen. Entweder man leugnet schlichtweg die Notwendigkeit OOP in Forth zu realisieren, stellt sich also auf den Standpunkt dass Objekte in einer minimalistischen Programmiersprache nichts zu suchen haben, oder man gesteht sich selber ein, der Aufgabe nicht gewachsen zu sein und muss notgedrungen auf andere Programmiersprachen ausweichen, will man mit Klassen arbeiten. Das ganze ist kein Nachteil von Forth, sondern hat etwas mit der Forth Community zu tun, also wie gut Forth verstanden wurde.

Auch mein eigenes Fachwisse ist in dieser Hinsicht nicht ausreichend. Ich kann also an dieser Stelle keine Forth OOP Extension anbieten. Das einzige was ich erläutern könnte wäre, wie man in Forth einen Array befüllt und wieder ausließt, aber das ist nur ein kleiner Teil der benötigten Funktionalität. Was ich jedoch sagen kann ist, dass OOP absolut notwendig ist will man größere Programme schreiben. Es ist nicht möglich und sinnvoll auf Objekte zu verzichten und stattdessen nur mit Methoden zu arbeiten. Ich habe das an einem eigenen Projekt ausprobiert. Obwohl der Sourcecode mit 500 Zeilen noch sehr überschaubar war, war der Verzicht auf Klassen eine Produktititätsbremse. Für den Computer sind solche Programme kein Problem. Ja ohne Klassen läuft der Code sogar effizienter als mit. Nur, irgendwer muss den Code schreiben und debuggen und das ist ohne OOP nur sehr umständlich möglich.

Ich würde mal schätzen dass in einigen Jahren unter Forth objektorientiertes Programmieren Einzug hält. Das heißt irgendwer wird sich schon finden, der den nötigen Handlungsdruck verspürt und eine Erweiterung in die comp.lang.forth Newsgruppe postet die ausreichend Funktionalität bereitstellt. Historisch gesehen dürfte die fehlende OOP Funktionalität mit ein Grund gewesen sein, warum sich als Alternative zu Fortran damals nicht Forth sondern C++ durchgesetzt hat.

Ein wenig weiter ist da schon die Sprache Factor. Diese erinnert an Forth, beinhaltet aber standardmäßig object-oriented programming. Leider ist Factor jedoch nicht so maschinennah wie Forth, sondern ist Applikationsorientiert.

2.3 Forth vs. C++

Ein Vergleich zwischen Forth und C++ fällt eindeutig zugunsten von Forth aus. Forth ist diejenige Sprache welche effizient ist, C++ ist es nicht. Forth ist leicht zu lernen, C++ nicht. Forth kann man auf Microcontroller portieren, bei C++ geht das nicht. Schauen wir uns als abschreckendes Beispiel ein Hello World Programm in C++ an, dann sieht man auf den ersten Blick dass C++ nichts taugt. Es werden dort Klassen angelegt, was bei Forth aus gutem Grund nicht zum Funktionsumfang gehört. Desweiteren benötigt man relativ viele Lines of Code um simple Aufgaben zu lösen. Man kann den Code entweder mit "g++ hello.cpp" übersetzen oder wer Wert auf hohe Performance legt kann auch den "-O3" Schalter aktivieren. Dadurch wird die heavy-loop in der Sleep Methode beschleunigt ausgeführt.

```
#include <iostream>
using namespace std;
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
    void sleep();
};
void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}
void Rectangle::sleep () {
  cout << "\n";
  for (int i=0; i<10;i++) {
    cout << i << "\n";
    for (int k=0; k<1000000000;k++) {
      int temp=2+2;
    }
  }
}
int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  rect.sleep();
  return 0;
}
```

Abbildung 2: Listing C++

Nur, an ein gut programmiertes Forth kommt der C++ Sourcecode nicht heran. Ursache dafür ist, dass der Code vorher noch durch den Gnu C++ Compiler übersetzt wird, was bedeutet dass man die Hilfe weiterer Werkzeuge in Anspruch nimmt und sich ohne Not abhängig macht nicht nur von der x86 Architektur sondern auch von ineffizienten Compilern. Dadurch geht der Purismus den Forth Liebhaber pflegen verloren. Dem Code fehlt jede Ausdruckskraft, es ist nicht empfehlenswert darin längere Programme zu schreiben.

Schaut man sich einige C++ Beispiel-Spiele an die auf github gehostet sind so stellt man fest, dass die Programmierer keinen einheitlichen Programmierstil verwenden, sondern häufig eine Mischung aus C und C++ schreiben. Fast schon zur Gewohnheit geworden ist es, auf das einleitende class statement zu verzichten und die C-Routinen einfach untereinander zu schreiben. Das mögen die Coder cool finden erschwert jedoch das Reverse Engineering wenn man mit cpp2dia und anderen Tools aus dem Code wieder ein UML Diagramm erzeugen will. Desweiteren hat sich bei C++ Programmierern die Unsitte einbürgert, Pointer auf sehr flexible Weise zu verwenden, offenbar gibt es auch keinen Standard, sondern jeder baut sich das so wie er es möchte. Das einzige Kriterium scheint ein bugfreies Programm zu sein, also dass sich der Code in Maschinencode übersetzen lässt. Nun mag man darüber hinwegsehen und diese Freiheiten den Programmierern zugestehen, wenn zumindest die abgelieferten Spiele von hoher Güte wären, doch das sind sie üblicherweise nicht. C++ Spiele sind grundsätzlich langweilig. Immer geht es um irgendwelche Raumschiffe die durch Asteroiden Gürtel fliegen und der Spieler muss auf alles ballern was sich bewegt. Wären wir noch Anfang der 1980'er könnte man die Grafik loben, dass jemand überhaupt ein Computerspiel programmiert hat und das butteweiche Scrolling. Doch im Jahhr 2017 interessiert sich dafür niemand mehr. Die Dinge auf die C++ Programmier besonders acht geben ist bedeutungslos geworden. Worauf es heute ankommt sind eher wissenschaftlich ausgerichtete Projekte, wo also der Sourcecode ein vorhandenes Paper praktisch realisiert und was dazu dient damit andere die Ideen weiterentwickeln können. Das ist jedoch bei C++ und dessen Community nicht der Fall.

2.4 Forth vs. Brainfuck

Der Vergleich Forth vs C++ fiel noch ganz eindeutig für Forth aus, C++ kann einfach mit dieser minmalistischen Programmiersprache nicht mithalten, was Eleganz, Lesbarkeit des Code und Komfort angeht. Wenn man jedoch einen Vergleich Forth vs. Brainfuck ansetzt sieht es schon nichtmehr so eindeutig aus. Brainfuck ist ähnlich wie Forth eine minimalistische Programmiersprache, es gibt sogar FPGA Implementierungen die das ganze als IP Core realisieren. Das heißt, Brainfuck ist dann Instruction Set und Programmiersprache in einem. Auf den ersten Blick besitzt Brainfuck gegenüber Forth den Nachteil dass es keine Wörter anlegen kann. Aber, wenn man einen "macro preprocessor" verwendet kann man sich Schlüsselwörter wie if, dup, und add definieren und damit dann richtig programimeren. Das heißt, man gibt dup ein, und der Makroprozessor erzeugt daraus dann den Brainfuck Code.

Brainfuck wird machmal mit APL verglichen. Es ist ebenfalls eine Minimal-Programmiersprache, und manche gehen soweit es als Esoterische Sprache zu bezeichnen. Forth hingegen gilt nicht als Esoterische Joke Sprache und verfolgt anders als Brainfuck höhere Ziele. Will also ernstgenommen werden. Insofern muss man sagen, dass auch erneut Forth der klare Gewinner ist. Forth ist eine richtige Sprache, während Brainfuck nur lustig ist.

2.5 ABI-CODE

Unter ABI wird das "Application binary interface" verstanden. Ein Standard, über den man Assembler Befehle unterschiedlicher Prozessoren vereinheitlicht. ABI-CODE bedeutet, dass man die x86 und AMD64 Iinstruction Sets an die Forth Sprache bindet. Vergleichbar mit dem SWIG Interface bei Java. Der Sinn ist es, nicht einfach einen weiteren Compiler zu schreiben, der Sprache A nach Sprache B transformiert, sondern es geht zunächst einmal zu definieren, dass Opcode #1 auf einer 32 bit CPU das selbe bedeutet wie Opcode #33 auf einer anderen 16 Bit CPU.

ABI-CODE und die darauf aufbauende Idee eines portablen Assemblers, gemeint ist nicht C sondern eher Dinge wie C– und PAF, dass also ABI-CODE verwendet wird als Backend für eine virtuelle Maschine. Der Ablauf könnte so aussehen:

Python Sourcecode -> Python Bytecode -> ABI-CODE -> Forth-CPU

Man hat darüber die komplette Pipeline abgebildet von einer objektorientierten Hochsprache in der man extrem effizient programmieren kann bis hin zu einer Lowlevel CPU die im Beispiel einer GA144 CPU nur eine Stackmaschine darstellt, also als extrem minimalistisch gilt.

2.6 Nachteile von Forth

Es ist relativ schwer, Nachteile von Forth zu entdecken aber es gibt sie. Der Flaschenhals von Forth sind seine Programmierer. Irgendjemand muss den Sourcecode editieren und debuggen. Dabei kommt es zu den gleichen Herausforderungen wie bei allen anderen Programmiersprachen auch. Das heißt in der Praxis dass auch der durchschnittliche Forthprogrammierer nur 10 Lines of Code pro Tag erstellen kann. Da bei Forth die Wörter um Platz zu spuren und besonders effizienten Code zu suggerieren häufig nebeneinanderweg geschrieben werden und nicht untereinander, dürfte die netto Zahl an Codezeilen sogar noch kleiner sein. Das diese Schätzung auf validen Daten beruht kann man bei einigen größeren Forth Projekten sehen. Die NASA hat Statistiken veröffentlicht wonach bei einem ihrer Forth Projekte pro Stunde 2 Zeilen Code in Forth entstanden sind (pro Kopf), und manchmal finden sich in den Implementierungen zu Virtual Maschines auch Angaben über den Umfang des Projektes. Durchschnittliche Forth Projekte bringen es auf 500 bis 10000 Lines of

Code [2].Die Schwierigkeit besteht darin, dass Forth wie jede andere Sprache auch manuell programmiert wird und das der Schwierigkeitsgrad zusätzlich noch erhöht ist.

Der Begriff “Code Library” wird bei Forth mit einer anderen Bedeutung verwendet. Normalerweise stellt man sich darunter größere Coderepositorien mit hundert Megabyte an Sourcecode vor. In der Arrayforth Implementierung heißt Code Library, dass Befehle wie Swap, Jump und Dup bereits implementiert sind. Das heißt, die Codelibrary besteht aus weniger als 1000 Bytes. Auch das Forth Repository Forth.net [5] ist im Umfang deutlich reduziert. Es handelt sich um nicht mehr als 100 kb in einer gepackten zip Datei wo Basis Funktionen aus der Mathematik und kleinere Forth Extension enthalten sind. Man sollte nicht den Fehler machen zu glauben, dass man in Forth programmieren können wie in C# und das es Bibliotheken gibt um die Programmierung weiter zu erleichtern, sondern Forth ist eine lowlevel Sprache, sie wurde entwickelt um auf Sub-Assembler Ebene mit der Hardware zu interagieren.

2.7 Hochsprachen nach Forth konvertieren

Wer wissen möchte wie man C Sourcecode nach Forth konvertiert ist damit nicht allein. in der offiziellen Fort FAQ taucht diese Frage ganz weit oben auf, so oft wird sie gestellt. Leider ist die Antwort ernüchtetnd. Es geht nicht. Es gab zwar einige Versuche, aber das Ergebnis ist nicht besonders gut. Auch der Versuch, Java Bytecode nach Forth zu übersetzen scheitert an Inkompiblitäten. Der Grund dafür ist, dass Forth ein minimalistsches System ist während die übrige Informatik einem Big is beautifull Schema huldigt.

Doch gehen wir das Thema from scratch an. Eine virtuelle Maschine lässt sich halbwegs einfach in C programmieren. Also eine Computersimulation einer Registermachine an die man Befehle sendet wie “LDA #1” und die dann ausgeführt werden. Solche Virtuellen Maschinen kann man auch als Stackmaschine aufbauen, wozu es zwar wenig Literatur gibt, aber es ist grundsätzlich möglich. Der nächste Schritt besteht dann darin, einen Hochsprachencompiler für diese Virtuelle Maschine zu schreiben, also einen Konverter der normalen C Sourcecode auf diese Maschine übersetzt. Üblicherweise geht dies mit Bison/yacc. Man parst zuerst den Sourcecode, erstellt daraus einen Abstract Syntax Tree und wandelt diesen dann in assemblerbefehle um, die werden zum Schluss noch in Maschinenopcodes verwandelt welche man dann auf der virtuellen Maschine ausführt.

Die Lehrbücher der Informatik erläutern üblicherweise das Programmieren eines Compilers für Registermachinen aber rein theoretisch ist auch ein Compiler für eine Stackmaschine vorstellbar. Nur das Problem ist, dass der damit erzeugte Code niemals optimal ist. In der klasschen Informatik akzeptiert man dies, obwohl bekannt ist dass beim Umwandeln von C nach Maschinencode Performance verloren geht, nimmt man das in Kauf, weil man in einer Hochsprache leichter programmieren kann.

Versuchen wir das jetzt auf die Forth Welt zu übertragen. Aus Sicht der Forth Puristen kommen Compiler nach Forth nicht in Frage. Sie würden immer nur suboptimalen Code erzeugen, also Code der durch Software generiert wurde, nicht jedoch manuell geschrieben wurde. Und irgendwie macht es auch keinen Sinn wenn man zuerst eine Minimalistische Programmiersprache entwirft, nur um darauf dann suboptimalen Code auszuführen. Die einzig richtige Antwort darauf lautet, dass man Forth am besten manuell zu Fuß schreibt.

Man kann sich eine Forth CPU also wie einen Koprozessor vorstellen. Er ist nicht dazu gedacht eine Standard-CPU zu ersetzen sondern wird verwendet um rechenintensive Aufgaben auszulagern. Will man für eine Forth CPU ein Programm schreiben, muss man es erst in einer Hochsprache Prototypen. Am besten eignet sich Python dazu. Und wenn die Software fehlerfrei läuft überträgt man einen Teil des Codes nach Forth.

Obwohl sich das zunächst wie unnötiger Mehraufwand anhört ist so aufwendig der Vorgang nicht. Wenn der Algorithmus erstmal spezifiziert wurde, ist es relativ einfach diesen in Forth nachzuprogrammieren. Man weiß ja bereits, welche Subfunktionen benötigt werden. Das heißt, man kann in Forth selber die Lösung eines Problems nicht finden. Dazu fehlt es bei Forth an zahlreichen Bequemlichkeiten wie objektorientierter Programmierung, Grafikausgabe und leichter Syntax. Man kann jedoch Forth als Zweitsprache nutzen, um ein Primzahlsieb in tausendfacher Beschleunigung zu berechnen.

Vermutlich wurde mit diesem Hintergedanken auch die GA144 entwickelt. Nicht als Desktop-Ersatz um damit im Internet zu surfen, sondern als High-Performance-Computer wenn man viel Rechenleistung benötigt. Forth Programmierung hat große Ähnlichkeit als wenn man eine Grafikkarte programmiert. Es wird das bereits erstellte Programm lediglich auf die Hardware angepasst.

Finite State Machine Forth darf man nicht mit einem Desktop PC verwecheln, wo man ein Betriebssystem hat und objektorientiert entwickelt. Sondern man muss sich auf Forth einlassen, das heißt es so verwenden wie es gedacht ist. Damit ist gemeint, dass man formale Methoden insbesondere Finite-State-Machines anwendet. Auf seinem Desktop PC wo windows, Linux oder was auch immer drauf ist, kann man schicke GUIs zusammenklicken, oder sich Algorithmen ausdenken, aber wenn man diese dann auf konkreter Hardware implementieren will kommt Forth ins Spiel. Hat man seinen Algorithmus so klar spezifziziert, dass sein State-Diagramm auf eine DIN A4 Seite draufgeht, dann ist der Zeitpunkt gekommen für eine Portierung nach Forth. Das heißt, es gibt für Forth keine Codegeneratoren und auch keine Converter aus anderen Sprachen, sondern die Best-Practice Methode ist nochmal bei Null anzufangen. Also ein komplett neues Forth Programm zu schreiben, die benötigten Subroutinen zu definieren und genau den Algorithmus implementieren den man vorher im Simulator getestet hat.

Und hier wird deutlich, was mögliche schlechtere Alternativen zu Forth wären. Eine Methode besteht darin, den Microcontroller in Assembler zu programmieren. Das hat jedoch den Nachteil, dass man es für jeden Microcontroller einzeln machen muss, weil sich die Opcodes ändern. Eine zweite Alternative ist die Verwendung eines C-Compilers. Aber ein C-Compiler erzeugt jedesmal ebenfalls anderen Code und man hat dort nicht die volle Kontrolle über das Ergebnis. Forth hingegen macht nur das, was man explizit programmiert hat. Das ist ideal geeignet wenn man das Bios modifiziert oder kryptographische Algorithmen implementieren möchte.

Kommen wir an dieser Stelle zu einem anderen Thema: Forth & Robotik. In vielen Papern ist zu lesen, dass mit Forth Roboter programmiert wurden. Man darf diese Behauptung nicht wörtlich nehmen. Ja ich würde sogar behaupten, dass es ausgechlossen ist, dass jemand nur mit Forth einen Roboterprototypen entwickelt hat oder einen biped Controller darin programmiert hat. Sondern der Ablauf in der Praxis dürfte eher so sein, dass man 10 Jahre lang mit Windows einen Roboter-Controller in der Simulation optimiert hat und dann als Sahnehäupchen die Finte-State-Machine in sauberem Forth nachprogrammert hat und die dann auf dem Microcontroller ausführt. Also bereits getesteten Code, der die nächsten 20 Jahre unverändert bleibt. Forth als ROM Chip sozusagen.

Ein gutes Beispiel dürfte die Speech-Synthese in Turboforth sein. Turboforth ist ein Dialekt, der fest in einem Taschenrechner als ROM eingebaut ist. Die Sprachsynthese ist nicht veränderbar, man kann sie lediglich aufrufen. Sie wurde mit Sicherheit nicht unter Forth programmiert, sondern wurde vermutlich in einem Tonstudio generiert, auf einem Desktop PC und dann lediglich in Turboforth bereitgestellt.

[5] https://theforth.net/

Finite State Machine Mit den obigen Ausführungen lässt sich sagen, wie genau in Forth programmiert wird. Man muss den Algorithmus den man implementieren möchte, schon als Finite State Machine vorliegen haben. Also in einer extrem reduzierten Form, wo man sehr genau weiß wie die Lösung aussieht. Diese Lösung kann man dann fast 1:1 in Forth programmieren. Anstatt der States definiert man Words, und probiert dann ein wenig mit dem Stack herum. Am Ende sollte der Algorithmus so laufen wie geplant. Wenn Probleme auftreten, dann lediglich in Detailfragen, also wie Forth einen Float nach Integer konvertiert oder ähnliches.

Kommen wir jetzt zum Anti-Forth Pattern. Man hat noch gar kein State-Diagramm, und es liegt auch kein Pseudocode in der Sprache Python vor. Jetzt ist Forth leider die falsche Sprache. Sondern hier muss man erstmal sich recherchieren, also sich überlegen wie der Algorithmus aussieht und einige Prototypen in Python schreiben. _Also klassische Informatik ohne Forth durchführen. Die Forth Sprache mit seinen lowlevel Fähigkeiten hilft nicht im geringsten dabei Algorithmen zu erfinden oder größerere Prototypen zu testen. Forth ist eher der letzte Schritt in einem Software-Engineering Projekt. Wo man also über Jahre hinweg die mathematische Literatur ausgewertet hat, und Programme geschrieben hat und jetzt die fertige Lösung als Forth Sourcecode einprogrammiert.

2.8 Wie gut ist das Software-Engineering bei Forth Projekten?

Auf Hardwareebene kann man getrost behaupten, dass die Forth Community Ahnung hat. Nicht nur, dass die Forth Chips sehr effizient sind, sondern auch Forth als maschinennahe Bytecode Sprache ist unübertroffen. Eigentlich müsste man die Java Virtual Engine mit Forth Code bestücken, sie wäre dann um einiges leistungsfähiger.

So sehr man auch die Leistungen von Forth im Bereich machinennahe Programmierung würdigen muss so wenig vermag Forth eine Antwort auf die Themen Robotik und Künstliche Intelligenz geben. Es ist zwar richtig, dass in den 1980'er Jahren einige Robotik-Projekte mit Forth realisiert wurden, aber diese unterscheiden sich nicht stark als wenn man C einsetzt. Bei Künstlicher Intelligenz geht es selten um die Performance der Hardware sondern ganz überwiegend stehen Software-Engineering Fragen im Mittelpunkt. Und hier macht Forth leider keine rühmliche Figur. Die Sprache selber unterstützt kein objektorientiertes Programmmieren, was es erschwert komplexere Anwendungen zu schreiben. Und minimalistische Robotersoftware zu schreiben ist für die Programmierer zu komplex. EIn Roboter ist etwas anderes als ein Microchip. Um einen Roboter zu programmieren muss man viel Domänenwissen in Computercode unterbringen. Schaut man sich real durchgeführten Forth Projekte aus dieser Richtung einmal an, so waren sie nicht besonders erfolgreich darin.

In der Theorie her kann man mit Sicherheit ein Roboter-Steuerungsprogramm in 50 Kilobyte an Forth Sourcecode unterbringen. Mehr braucht es vermutlich nicht um eine inverse Kinematik, Motion Primitive und ein wenig Bildverarbeitung zu betreiben. Das Problem liegt eher auf der Programmiererseite. Ein solches Programm schreibt sich nicht von allein, jemand muss sich die Behavior Trees ausdenken, die Sache debuggen usw. Das geht mit Forth leider nicht so gut. Wenn man schon eine umfangreiche Robotersoftware in Java geschrieben hat kann man diese garantiert auf Forth portieren. Dabei würde man die wichtigssten Algorithmus auf ihre Kernbestandteile reduzieren und die vielen Hilfsroutinen die bei der Entwicklung benötigt wurden einfach weglassen. Nur, dazu muss man den umfangreichen Java Code erstmal haben. Und genau hier liegt das Problem. Derzeit ist außerhalb von Forth nur wenig positives zu berichten.

Ich will mal versuchen ausführlicher zu erläutern, warum Forth bei der Robotik Programmierung keine Hilfe darstellt. Üblicherweise wird ein Robot-Control-System bottom up programmiert. Es treten dort viele kleine Probleme die nacheinander gelöst werden müssen. Beispielsweise muss benötigt man die Formel mit der man ein Auto auf einen Waypoint hinlenken kann. Die Formel dafür findet sich meist bei Stackoverflow oder man findet sie selber durch herumprobieren aus. Mit programmieren im eigentlichen Sinn hat das nichts zu tun, eher damit dass man in einer Tabellenkalkulation die Werte einträgt und damit herumspielt. Das eigentliche Programmieren besteht darin, dass man die Formel dann als Sourcecode aufschreibt. Selbstverständlich kann man dies in Forth tun, aber jede andere Sprache wie Python und Co eignet sich dafür genauso gut. Die meiste geht also nicht dabei dabei drauf, dass man im Speicher mit Pointern hantiert oder Code schreibt, sondern damit dass man Stackoverflow durchsucht, mit dem Roboter herumprobiert und solche Dinge. Dabei kann Forth leider null helfen.

2.9 Parallelentwicklung in Python und Forth

Wenn man Forth einsetzt ergibt sich die Frage woher man den Sourcecode bekommen möchte. Standardmäßig wird bei einer Forth Engine exakt null ausführbarer Code mitgeliefert, auch keine Libraries oder gar Robot-Control-Systeme. Die Antwort lautet, dass man parellel zu Forth noch in einer anderen Sprache programmiert. Die best-practice Methode sieht so aus, dass man zuerst in Python einen 2D Simulator für ein Micromouse Labyrinth schreibt, darin dann ebenfalls in Python einen Behavior Tree für einen autonomen Agenten entwickelt und die die Software läuft extrahiert man daraus stripdown die wichtigsten Codeteile und portiert sie nach Forth. Für den Simulator mitsamt KI wird man 2000 Zeilen Code in Python benötigen vermutlich eher mehr, wenn der Roboter in den Kurven ein ausdriften vermeiden soll. Und daraus kann man dann die wenigen wichtigen Stellen herausnehmen. Vielleicht 20 Funktionen und die sehr kompakt in Forth Code schreiben. So dass man am Ende eine Minidatei mit 2 kb hat die man auf den echten Micromouse Roboter draufflasht.

Der Vorteil ist, dass der Microcontroller ruhig "c-compiler unfriendly" sein darf, weil Forth bekanntlich überall läuft. Wenn es Bug auftritt fixt man den zuerst im Python Simulator und passt anschließend den Forth Code an. Was zunächst einmal wie eine mühsame Doppelentwicklung klingt ist erstaunlich effizient. Wenn man bereits ausführbaren Python Code besitzt für das Steering und die Pfadplannung lässt sich dieser relativ unkompilizert in Forth nochmal neuschreiben. Theoretisch kann man den Python Code auch nochmal in Assembly neu formulieren, nur Assembler ändert sich bei jedem Microcontroller. Man wäre dann auf eine Plattform festgelegt. Und ebenfalls rein theoretisch kann man den Python Code auch nach C übertragen und von dort nach Assembler übersetzen. Nur, ob man den Code nach C oder nach Forth konvertiert macht keinen großen Unterschied. Und Forth hat den Vorteil dass klarerer wird um was es geht: eine minimale Programmdatei mit ausführbarem Code.

Was jedoch nicht funktioniert ist wenn man auf den Prototypen in Python verzichtet, also gar keine 2D Simualtion nutzt und gleich bare metal die Pfadplannungsalgorithmen in Forth formuliert. Der Schwierigkeitsgrad ist zu hoch, das muss scheitern. Wobei das nicht nur bei Forth scheitert, sondern nicht viel anders in C geht. Wer versucht direkt einen Roboter in C zu programmieren ohne vorher im 2D Simulator verschiedene Algorithmen ausprobiert zu haben wird ebenfalls sehen, dass es nicht geht. Es ist eben keine Kleinigkeit eine Micromouse durchs Labyrinth zu schicken.

C vs Forth Forth hat gegenüber C aber auch Nachteile. Wenn der Algorithmus sehr kompliziert ist und Arrays verwendet ist es leichter ihn in C zu programmieren und nach Machine Code zu compilieren. Desweiteren gilt normales C bereits als Anspruchsvoll in der Bedi-

enung und Forth ist vom Schwierigkeitsgrad noch höher. Wer einfach nur einen funktionierenden Roboter haben will ist kann sogar auf C und Forth komplett verzichten und stattdessen einen Microcontroller nutzen auf dem Python läuft. Er wird dann noch schneller den Algorithmus darauf portieren.

2.10 Definition einer Toylanguage

Der Begriff Toylanguage wird manchmal verwendet um auszudrücken dass man eine Sprache nicht mag. So behaupten C++ Hater, dass es sich dabei um eine Toylanguage handeln würde. Doch man kann auch objektive Kriterien anführen und zwar das Vorhandensein von Softwarebibliotheken. Unzweifelhaft besitzt C++ eine sehr große Anzahl von Librarys ist also per Definition keine Toylanguage. Das ich persönlich C++ nicht mag ist dabei egal. Fakt ist jedenfalls dass die Sprache intensiv eingesetzt wird und dass in den Librarys wiederkehrende Probleme gelöst wurden.

Als Toylanguage par excellence kann man hingegen Brainfuck bezeichnen. Dort gibt es nur sehr wenige Libraries. Das heißt, Brainfuck wird nicht verwendet um darin effektiv Programme zu schreiben sondern Brainfuck steht selbst im Fokus. Es ist also eine akademische Sprache. Und hier ist der Übergang zu Forth gegeben, auch zu Forth gibt es keine Libraries, jedenfalls keine umfangreichen. Mit etwas suchen stößt man auf “The forth net” welche aber nur dokumentarischen Charakter besitzt und allein vom Umfang her sehr klein ist. Demnach ist auch Forth eine Toylanguage. Natürlich kann man Forth in eine professionelle Sprache verwandeln und dafür Libraries schreiben, doch derzeit ist das nicht der Fall. Offenbar gibt es also zuwenig Leute die das versuchen.

Auch die anderen Programmiersprachen kann man anhand der Fülle vorhandener Bibliotheken in Toylanguage oder echte Sprache untergliedern. Die bekannten großen Sprachen wie Java, C#, C und Python sind über jeden Zweifel erhaben. Sie besitzen mehr als 1000 Librarys, teilweise sogar mehr als 100000. Das ist Ausdruck von sozialen Einschätzungen. Das heißt, wie gut eine Sprache von den Programmierern angenommen wird und wieviel Energie in den Aufbau von Bibliotheken gesteckt wird. Will man nichts verkehrt machen sucht man sich einfach jene Sprache welche die meisten Bibliotheken besitzt und hat damit eine Sprache ausgewählt, die im Produktiveinsatz verwendet wird und nicht nur von akademischem Interesse ist.

Wie man an diesem kleinen Exkurs gesehen hat, erfüllen Programmiersprachen überwiegend einen sozialen Zweck. Sie werden von Menschen verwendet, weil sie ein Problem lösen wollen oder programmieren lernen möchten. Eine Programmiersprache ist nur so gut, wie die Community dazu. Doch kommen wir zu Forth. Forth ist eine Sprache ohne Library. Ähnlich wie Brainfuck ist die Sprache intellektuell interessant aber es fehlt an einer Community die sie freiwillig spricht. Die Anzahl der Paper die sich mit Forth beschäftigen ist weitaus höher als die tatsächlich durchgeführten Projekte in Forth. Woran das liegt ist simpel: Forth wurde auf eine leichter Interpretierbarkeit durch Interpreter entwickelt. Forth ist ähnlich leicht zu parsen wie Brainfuck. Sie orientiert sich an Maschinen nicht aber an Menschen.

3 Virtualisierung

3.1 Forth ist eine Virtuelle Maschine

Damit Forth Befehle auf einer Standard-CPU ausgeführt werden können benötigt man eine virtuelle Maschine. Also ein Computerprogramm was Forth Code in Maschinenopcodes des Prozessor übersetzt. Eine solche Virtuelle Maschine wird als Forth System bezeichnet. Üblicherweise sind diese Maschinen in Assembler programmiert und auf die jeweilige CPU angepasst. In dem NEXT Befehl ist der Kern dieser VM verborgen. NEXT bedeutet, dass der aktuelle Forth Befehl auf der Hardware ausgeführt wird. Will man auf eine Virtuelle Maschine muss man Forth direkt in Hardware ausführen, dazu benötigt man einen dezidierten Forth Prozessor.

Wie man vielleicht schon ahnt gibt es dieses Konzept nicht nur bei Forth. Auch pyCPU (Python Hardware Processsor). Wiedereinmal mehr ist Forth seiner Zeit also weit voraus.

3.2 Forth is dead? Ja

Vor einiger Zeit fragte Paul Frenger in einem Text der in einer ACM Fachzeitschrift erschien ob Forth nicht längst überholt sei [6]. Die Antwort auf diese ernstgemeinte Frage lautet, dass Forth in der Tat obsolet ist, es gibt längst bessere Scripting Sprachen welche Forth ersetzen können. Eine davon ist Javascript. Für Javascript gibt es inzwischen ausgezeichnete x86 Emulatoren[6] womit man die x86 CPU in einem Browserfenster simulieren kann und darin dann MS-DOS, Windows 1.0 oder was auch immer startet. Die Geschwindigkeit solcher Emulatoren, die es übrigens auch für die MOS6502 CPU sind ist ausgesprochen gut. Man kann also sagen, dass sich mit geschickter Programmierung zu Forth eine Alternative bietet.

3.3 CPU Simulator in Forth?

ich habe jetzt mehrfach versucht, Teile meiner eigenen Robotics-Library nach Forth zu portieren. Eigentlich eine simple Aufgabe, weil der Code vorher von mir selbst in Python geschrieben wurde und es leicht sein müsste ihn in die Python Syntax zu überführen. Aber es hat nicht funktioniert. Zwei kleine Funktionen zur Abstandsberechnung zwischen zwei Punkten habe ich hinbekommen. Der Zeitaufwand dafür war imens. Und wenn man komplexere Funktionen die Array Manipulationen benötigen nach Forth übertragen will, dürfte es vermutlich Jahre dauern bis der Code getestet ist. Es verwundert daher nicht, dass es für Forth fast keine Beispielprogramme gibt. Die Sprache ist unprogrammierbar. Dagegen ist Assembler ja noch als leicht zu bezeichnen.

Es stellt sich also die Frage was man mit Forth anfangen kann. Einfach ignorieren wäre eine Methode. Spannend wäre es, wenn es aber doch irgendwie gelingt dafür Programme zu schreiben. Eine Möglichkeit wäre die CPU Emulation. Zur Erinnerung: heutige C-Compiler setzen einen Instruction Set voraus und zwar den AMD64 Standard. Dieser zeichnet sich dadurch aus, dass es relativ viele Befehle gibt und mindestens 30 Register (besser mehr) verfügbar sind. Dafür optimiert der C-Compiler dann den Code. Ein Forth Chip stellt diesen Instruction Set nicht zur Verfügung, also kann man auch keine C-Compiler nutzen um dafür Progrmame zu compilieren. Es gibt jetzt zwei Optionen: entweder verbessert man die C-Compiler, so dass sie als Ausgabe Forth Code erzeugen, oder man schreibt in Forth einen x86 Simulator, so dass man in Forth AMD64 Code ausführen kann.

Aber bevor man sich an einen CPU Simulator in Forth wagt kann man erstmal mit Python sein Glück versuchen. Es gibt mehrere – halbwegs leicht verständliche – Projekte wie z.B. py65 was eine MOS 6502 CPU in Python nachbildet. Von der Funktion vergleichbar mit qemu nur eben sehr viel kompakter. Nicht die Geschwindigkeit steht im Vordergrund sondern dass der Code schön leicht verständlich in Python programmiert wurde. Und an diesen Python Code kann man dann 6502 Assembler Befehle senden, so jedenfalls die Idee dahinter.

Theoretisch wäre es möglich, py65 nach Forth zu portieren und plötzlich könnte man dann in Forth einen C-64 booten. Dadurch wäre dann massenhaft Software die für den C-64 entwickelt wurde auch

[6] https://github.com/copy/v86

unter Forth lauffähig. Man könnte so ein Stückweit die Forth Softwarekrise entschärfen. Wenn ich mich recht entsinne habe ich irgendwo schonmal eine halbfertige Implementierung in Forth gesehen die aus 100 Lines of Code bestand. Das heißt, offenbar bin ich nicht der einzige der darüber nachgrübelt.

Gewissermaßen stellt Forth die letzte große Bastian da. Es ist vergleichbar mit einem schwer zu besteigenden Berg. Vielleicht gelingt es eines Tages, dorthin einen Weg zu schlagen.

4 Rekursion

4.1 Rekursive Künstliche Intelligenz

In der Künstlichen Intelligenz gibt es eine Subdisziplin namens rekursive selbst-modifizierende Algorithmen zur Erlangung von Super-Human-AI, abekürzt AIXI. Vereinfacht gesegt soll sich einerseits die Funktion selbst immer wieder aufrufen, aber dabei noch den Sourcecode modifizieren. Wenn man jetzt sowas explizit als Aufgabe formuliert, dann ist Forth die geeignete Programmiersprache dafür. Nicht nur dass Rekursive Aufrufe sehr gut unterstützt werden, sondern auch das Verändern des eigenen Codes zu Laufzeit ist ein Wesensmerkmal von Forth. Ob man soetwas realisieren kann ist unklar, aber es passt zumindest technisch ziemlich gut zusammen.

Unter dem Stichwort “Super-recursive Algorithm” wird anders als bei AIXI konkrete praktische Beispiele diskutiert. Also mathematische Verfahren wo selbst-lernende Turing-Maschinen angestrebt werden. Obwohl das nicht explizit unter dem Stichwort Forth Programming läuft, eignen sich Stackmaschinen jedoch ausgezeichnet dafür derartige Algorithmen praktisch umzusetzen.

Nochmal: Compiler für Stackmachinen Ausgangspunkt war die Überlegung, dass Registermaschinen als C-friendly gelten und Stackmachinen als C-unfriendly. Daraus ergibt sich das Problem wie man für Stackmachinen wie die GA144 Software schreiben soll. Ein C-Compiler gibt es dafür nicht und vermutlich wird auch nie einer erfunden. Also besteht der Ausweg darin, sich Probleme zu suchen, die dezidiert für Stackmaschinen geschaffen wurden. Und voila, man gelangt auf direktem Wege zu superrekursiven Problemen. Wo also die Studenten schon in der Aufgabenstellung dazu angehalten werden, Rekursion und einen Stack zu nutzen. Solche Probleme sind wie geschaffen für die Sprache Forth und für Forth CPUs.

Kennt man den Algorithmus kann man ihn fast als Pseudocode in den Forth Interpreter eintippen. Das spart Zeit und einen C-Compiler vermisst man auch nicht.

4.2 Grenzen der Rekursion

Wer die Arbeiten von Jürgen Schmidthuber kennt wird sich garantiert gefragt haben, ob die vorgeschlagene Künstliche Intelligenz realistisch ist und wie man sie in Forth implementiert. Genauer gesagt lautet die Frage ob man sehr kurzen aber umso raffenierteren Algoirthmen Intelligenz erzeugen kann. Also ein Programm konstruieren was mittels Stack arbeitet, sich selber aufruft, womöglich eigene Forth Words erzeugt und dabei ständig intelligenter wird und sich immer mehr dem Ziel anpasst. Die Antwort darauf lautet nein, die Suche danach ist vergleichbar als wenn Physiker versuchen ein Perpetum Mobile zu bauen. Beweisen lässt sich diese ernüchtene Feststellung nicht direkt, aber anhand von real durchgeführten Programmierprojekten spricht viel dafür. Schauen wir uns einmal typische Programme an, die nach dem Rekursions-Schema arbeiten und sich der Chaostheorie verpflichtet pfühlen. Die Gemeinsamkeit besteht darin, dass sie sehr kurz sind, im Sinne von wenig Zeilen Code. Und das unabhängig davon ob sie in Java oder Forth erstellt wurden. Auch der Area Filling Algorithmus den Elizabeth Rather in den 1980’er Jahren in einer Folge von Computer Chronicles vorgestellt hat war sehr kurz und mit sicherheit verwendete er Rekursion. Und was sich darüber sagen lässt ist, dass es nicht funktioniert.

Der Grund warum die klassische Informatik Rekursion als Hexenwerk und Forth Chips als Hölle bezeichnet ist nicht etwa weil sie uns davor warnen will weil es zu mächtig ist, sondern weil Rekursion eine schwache Technik ist. Damit ist gemeint, dass Superhuman -AI mit Sicherheit nicht in ein 10 kb großes Forth Programm hineinpasst was sich irgendwie im Speicher vermehrt, sondern echte Künstliche Intelligenz basiert auf einer Maßzahl aus der klassischen Informatik, gemeint sind Lines of Code. Will man die Komplexität also die Leistungsfähigkeit von Software erhöhen so muss man diesen Wert nach oben justieren. Dadurch stellt sich fast von allein soetwas wie Eleganz, Performance und Leistung ein.

Die Schwierigkeit mit Rekursiven Algorithmen und speziell superrekursiven Algoirthmen ist, dass sie wie der Name schon andeuten an einem Berechenbarkeitsmodell orientiert sind. Die Lösung also darin sehen, dass die CPU eine Aktion ausführt und zu weiteren Sprungadressen im Speicher springt. Das ist jedoch nicht die Lösung, sondern das ist die Basis um überhaupt eine Lösung in Erwägung zu ziehen.

Werfen wir nochmal einen Blick auf den C-Compiler welcher von der Forth Community manchmal belächelt wird. Die Sprache C besagt, dass die Hardware im Grunde egal ist. Es spielt keine Rolle ob man die Software für eine Stackmachine, eine virtuelle Maschine oder einen Supercomputer programmiert, das einzige was zählt ist der Sourcecode. Wie der ausgeführt wird ist Sache von anderen. Diese Überheblichkeit gegenüber Lowlevel CPU Architekturen ist symptomatisch für die Mainstream-Informatik und sie ist das stäarkere Argument. Damit ist gemeint, dass rekusrive Algorithmen, Stacks und Forth keine eingebaute Intelligenz besitzen, die man nur freischalten müsste, sondern das Software genau das ist, was man einprogrammiert hat. Und je mehr Codezeilen ein Programm hat, desto mehr Intelligenz ist darin enthalten.

Bei Mini-Programmen im Umfang von 10 kb welche Super-Intelligenz versprechen hat man immer den Eindruck, als ob eine sehr naive Vorstellung vorhanden ist. Nach dem Motto, dass die Turing-Maschine bereits intelligent sei, und man lediglich den Schlüssel in Form des Sourcecode bräuchte um diese Grundintelligenz zu aktivieren. So ähnlich als würde man Energie aus dem Vakuum schöpfen, wenn man weiß wie es geht. Das ist das genaue Gegenteil von Wissenschaft, das ist magisches Denken. Genau diesem Virus sind Jürgen Schmidhuber und andere aufgesessen und sie glauben, dass Rekursion und Stacks der Schlüssel sind.

Quellen Vielleicht fragt mich der eine oder andere, wo denn meine Quellen für diese kühnen Thesen seien und wie ich das beweisen möchte. Meine Quellen sind nichts geringeres als die tatsächlich durchgeführten Roboterwettbewerbe. Wo also Aufgaben gestellt werden, die von Teams unter kontrollierten Bedingungen abgearbeitet werden müssen. Da jedes Team gewinnen will ist anzunehmen, dass solche Wettbewerbe ein Sammelbecken für effiziente Algorithmen sind. Das interessante ist, dass sich im laufe der Jahre jene Software durchgesetzt hat, die aus mehr Zeilen Code bestand und gleichzeitig dass noch nie rekursive stack-basierende Verfahren zur Anwendung kamen. Probiert wurde es, nur brachte es keinen Erfolg, Also hat man wieder Abstand genommen. Wer also glaubt, doch mit rekursiven Verfahren und Programmen die in Forth geschrieben wurden, bei solchen Wettbewerben gewinnen zu können, nur Mut, deine Mitarbeit ist erwünscht. Allerdings befindet man sich damit in der Defensive, in dem Sinne dass man das beweisen muss was sehr unwahrscheinlich ist.

Woran das genau liegt, dass man die Intelligenz eines Roboters

verdoppelt wenn man die Lines of Code des Projektes verdoppelt hingegen nicht der gleiche Erfolg eintritt wenn man sich bessere, rekursive Algorithmen überlegt ist nicht so ganz geklärt. Ich argumentiere hier auch nicht von einer theoretischen Perspektive sondern beshreibe die Wirklichkeit bottom up. Es ist eine Beobachtung. Man könnte es ungefähr so deuten, dass eine Maschine also ein Computer nach dem Einschalten Null Intelligenz besitzt und das jede Funktionalität die man benötigt mühsam einprogrammiert werden muss und das der Aufwand dafür steigt, je mehr Zeilen Code man erstellt.

Medium Computer Die C-Programmierer und die Forth Programmierer könnten unterschiedlich nicht sein. Wer in C programmiert hat akzeptiert dass der Computer nur ein Medium ist, ähnlich wie die Leinwand in einem Kino muss man sie bespielen und zwar mit selbst ausgedachter Software. Forth Programmierer hingegen hängen an der Idee, dass der Computer selbst das Medium wäre. Folglich sehen sie nicht Sourcecode als das wichtigste sondern sie fokussieren auf die Hardware des Rechners. Forth Programmierer glauben, der Film wäre schon da, man müsste nur genau hinschauen. Manchmal setzen sie auch spezielle Programme ein, die sich Artificial General Intelligenz nennen und was die Aufgabe hat, den imaginären Film freizuschalten.

Man kann sagen, dass Forth Programmierer sich einen naiven Zugang zum Computer bewahrt haben. Sie denken nicht, dass selbst programmierte Software der Schlüssel ist, sondern dass minimal Programme und gut implementierte Algorithmen besser geeignet sind für einen Zugang zur Maschine.

4.3 Codegenerator

In der traditionellen Informatik dient ein C-Compiler als Codegenerator. Der User gibt dort Pseudocode ein und erzeugt dann den passenden Maschiinencode. Da bei Forth kein C-Compiler funktioniert stellt sich die Frage nach möglichen alternativen Codegeneratoren. Die einfachte Möglichkeit ist, wenn man den Code einfach selber erstellt. Das heißt, zuerst erstellt man in den Code, testet ihn und wenn er funktioniert portiert man ihn in die Forth Syntax. Ein wenig besser ist es, wenn man das Forth so anpasst, dass es C-friendly wird, also Register emuliert, damit man leichter für diese Plattform Code erzeugen kann. Als dritte Möglichkeit kann man auch eine sehr forthtypische Lösungsstrategie verwenden. Diese besteht darin, die Aufgabenstellung komplett zu ignorieren und stattdessen die passende Aufgabe für den Hammer zu suchen den man besitzt. Eine passende Aufgabe wie gemacht für Forth hat etwas mit dem Stack zu tun und ist rekursiv formuliert. Und voila, dass ist aus Sicht von Forth die befriedigenste Antwort.

Man sucht sich also einen Programmierwettbewerb, bei dem die Teilnehmer ein Fraktal erzeugen lassen und dazu einen Stack verwenden sollen. Sowas lässt sich sehr gut in Forth implementieren. Man kann da schön ein Word definieren, was sich selber aufruft, und Variablen braucht man auch keine.

Passende Probleme für das vorhandene Werkzeug Die Ausführungen der Einleitung möchte ich etwas vertiefen. Als Codegenerator bezeichnet sich das VFX Forth Environment. Es handelt sich dabei um ein kommerzielles Paket mit dem man Forth Programme erstellen kann. Das läuft unter MS-Windows und es werden einige Beispiele mitgeliefert. Aber, die Erfinder von VFX Forth verstehen unter Codegenerator, dass ihr Forth in der Lage ist, aus Forth Statements Assemblerbefehle zu erzeugen. Das heißt, Forth Code soll auf einem Microcontroller ausgeführt werden. Woher dieser Forth Code stammen soll, verraten sie nicht. In den Beispielen geht es überwiegend um Windows spezifische Dinge, also das Erzeugen eines Fensters oder das Programmieren eines Webbrowsers. Das mag alles mit VFX Forth lauffähig sein, doch eigentlich ist gegenüber gforth kein echter Vorteil erkennbar.

Man kann diese Sache aber noch detailierter beschreiben. Im Grunde genommen ist VFX Forth eine Kopie einer C-Compilers. Nur eben nicht C->Assembly sondern Forth->Assembly. Der Programmierstil zum Entwickeln des Programms ist identisch. Böse formuliert wird also das Forth Programm nach den selben Methoden erstellt wie das C-Programm auch. Hier wurden die Vorteile von Forth nicht ausgenutzt, im Grunde hätte man auch einen normalen C-Compiler nehmen können. Damit geht die Entwicklung weitaus schneller. Nein, um Forth richtig auszulasten muss man sich andere Aufgaben suchen, also welche die nichts mit Windows Programmierung oder embedded zu tun haben, sondern aus dem bereits erwähnten Spektrum der rekursiven Stackmanipulation entstammen. Hier ist eine Aufgabensammlung mit Rekursionsproblemen [7] Eigentlich ist das eine Seite, womit die Studenten lernen sollen in Java zu programmieren, doch Forth eignet sich zur Problemlösung um einiges besser.

Und es kommt noch besser. Zu den Aufgaben ist dankenswerterweise auch die Lösung mit abgedruckt, natürlich ebenfalls in Java. Obwohl Java ähnlich wie C eigentlich eine Sprache ist, von der man in der Forth Community nicht viel hält, ist in diesem Fall das Beispiel in Java ausgezeichnet. In der zweitvorletzten Aufgabe soll man konkret Primzahlen rekursiv bestimmen. Die abgedruckte Java-Lösung ist extrem prägnant. Sie ist extrem kurz, nimmt zwei Paameter entgegen und lässt sich fast 1:1 in Forth nachprogrammieren. Die Parameter n und m legt man einfach auf den Stack, die Syntax passt man leicht an und fertig ist das perfekte Forth Program.

Offenbar geht es also nicht um Forth vs C sondern worum es geht ist die Art der Aufgabenstellung. Man muss ausschau halten nach rekursiven Aufgaben und wenn man dort in Java oder in C eine Lösung schon hat, kann man diese erstklassig für Forth nutzen. Ja, ich würde behaupten wollen, dass der Forth Sourcecode in diesem Fall sogar eleganter aussieht als die Java Lösung.

Aus diesem Beispiel ist etwas deutlich geworden. Man kann Forth so programmieren als wäre es C (prozedurale Programmierung), man kann Java so programmieren als wäre es Forth (rekursive Programmierung) oder man kann Forth wie Forth und C wie C programmieren. Also die Regel aufstellen dass für rekursive Aufgaben die einen Stackverwenden nicht Java verwendet werden darf (obwohl das ginge) und für prozedurale Aufgaben wo klassische loop algorithmen eingesetzt werden, nicht Forth benutzt werden darf, obwohl auch das geht.

Primzahlen rekursiv Hier [8] ist nochmal ein Stackoverflow thread bei dem mit Python rekursiv Primzahlen bestimmt werden sollen, die kürzeste Lösung ist die vorletzte, sie ist vergleichbar mit der oben zitierten Java Lösung. Der Sourcecode ist ganz eindeutig Python, ich habe mir erlaubt über Copy&Paste in die obige Abbildung zu überführen. Es ist ein sehr prägnanter Algorithmus und das entscheidene ist, dass er fast 1:1 nach Forth übertragen werden kann. Klar, Forth kennt keinen “def” Befehl und auch kein Print, aber das sind Detailfragen. Wichtiger ist hingegen, dass die Aufgabe “Finde eine Primzahl rekursiv und bei Verwendung eines Stacks” sehr gut auf die Möglichkeiten von Forth zugeschnitten ist. Es ist dasjenige Prinzip, wie man Forth richtig programmiert. Also die Sprache genau so einsetzt wie es gedacht war.

Rekursive Greifplanung Im letzten Abschnitt wurde herausgearbeitet, dass sich Forth ausgesprochen gut für Aufgaben eignet, die

[7] http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/aufgaben/rekursion.html

[8] https://stackoverflow.com/questions/37095508/how-do-i-find-a-prime-number-using-recursion-in-python

Algorithmus 1 Primzahl rekursiv

```
def is_prime(N, a=2):
    if N <= 1:
        return
    elif a >= N:
        print(N)
    elif N % a != 0:
        is_prime(N, a + 1)
is_prime(23)
```

etwas mit Rekursion und mit dem Stack zu tun haben. Es ist gewissermaßen ein Heimspiel. In der Robotik gibt es zahlreiche Verfahren die genau das fordern. So berichtet die Literatur über rekursive Greifplanungsalgorithmen. Wo also die Lösung darin besteht, dass sich eine Funktion sich selbst aufruft um so den Lösungsraum zu durchschreiten. Solche Strategien sind ausgesprochen Forth-friendly. Damit ist gemeint, dass sie sich nur schlecht in C aber umso besser auf einem GA144 Chipsatz implementieren lassen. Man muss dort nicht umdenken oder an die Maschine anpassen sondern man implementiert den Algorithmus genau so wie er gedacht war.

Um vielleicht beim früher erwähnten Vergleich mit dem Fixed Gear Fahrrad zu bleiben. Es ist so ähnlich als wenn man dem Fixed Gear Fahrrad nur dort fährt, wofür es konstruiert wurde. Auf der Innenseite einer Radrennbahn, wo also diese Fahrradkonstruktion nicht abenteuerlich ist, sondern wo sie im Reglement so drinsteht.

Ein weiteres Problem aus der Spieltheorie wäre ein rekursiver Backtracking Solver für Sudoko. Auch dieser lässt sich ausgezeichnet in Forth implementieren. Man kann ihn sogar zusätzlich noch parallelisieren und als Demonstration auf einem GA144 Chipsatz ausführen. Diese beiden Beispiele haben gezeigt, dass man Forth am leichtesten programmieren kann, wenn man sich spezifische auf Forth zugeschnittene Aufgaben sucht.

Ein etwas anspruchsvolleres Beispiel aus der Robotik wäre rekursive Natural Language understanding mit DeepLearning. .Es gibt dazu mindestens ein Video und ein Paper online was das Verfahren erläutert. Und wie sie lieber Leser vermutlich ahnen, eignet sich besonders eine Programmiersprache dafür sowas zu implementieren. Würde man das in Forth tun und in einer Forth Zeitschrift publizieren, wäre das nicht nur Ontopic, sondern wäre vielleicht sogar den ersten Preis erzielen, weil die Programmiermethode Forth-Style in Reinstform verkörpert.

4.4 Rekursiver Flood Fill Algorithmus

Wie die Forth Programmiersprache funktioniert ist bekannt, es gibt viele Implementierungen in C und Javascript. Ebenfalls bekannt ist wie Forth CPUs funktionieren, die J1 CPU gibt es zum kostenlosen Download im Internet. Unklar ist hingegen was man damit anfangen soll, genauer gesagt wie man in Forth richtig programmiert.

Eine Möglichkeit wäre es einen C to forth Compiler zu schreiben. Auf github konnte ich einen finden, der mit einer Bison Grammar funktioniert. Wahrscheinlich kann man mit Hilfe des LCC Compilern noch einen entwickeln der etwas effizienter funktioniert. Dadurch könnte man dann vorhandene C Programme auf eine Forth Stackmachine konvertieren. Die Frage ist nur: ist das schon richtiges Forth? Ausgeführt werden die Programme, aber es fühlt sich nicht richtig an. Nein, wenn man schon Forth verwendet sollte man es auch mit den passenden Algorithmen einsetzen. Ein sehr bekannter Algorithmus der sich inhaltlich an Forth anpasst ist der rekursive-stackbased flood fill Algorithmus. Ein Algoirthmus der exakt so funktioniert wie auch Forth funktioniert. Man kann ihn nahezu 1:1 in Forth Code implementieren.

Das interessante daran ist, dass man ihn nicht nur in Forth programmieren kann sondern auch in Standard Java. Der Sourcecode sieht dann – obwohl es Java ist – ebenfalls sehr nach Forth aus. Hier ist eine Beispielimplementierung [9] welche nicht länger ist als 50 Zeilen und in der Einleitung vor einer Stackoverflow Exception warnt. Schaut man sich den Code an, so ist das eindeutig Forth Style. Mit diesem Code könnte man auch Chuck Moore beeindrucken.

Offenbar geht es also nicht um die Programmiersprache Java vs. Forth sondern worum es geht ist die Art des Algorithmus. Wenn man eine rekusrive Stackbased Variante wählt ist das gleichbedeutend mit einem Forth-Style. Also eine Programmiermethode welche kompatibel ist zu einem Minimal Instruction Set Computer. Wenn man hingegen einen Floodfill Algorithmus auf andere Weise realisiert ist es ein Non-Forth Style.

Warum diese Unterscheidung wichtig ist liegt auf der Hand. Im Grunde braucht man sich nicht direkt mit Forth zu beschäftigen wenn man etwas über diese Programmiersprache lernen möchte. Sondern Forth ist nur eine Möglichkeit wie man rekursive Flood Fill Algorithmen implementiert. Genausogut kann man auch Python oder C verwenden. Worum es eigentlich geht das ist die Rekursion, also eine besondere Form der Programmierung die man auf einem Computer anwendet um Probleme zu lösen.

Und hierüber erhält man auch eine Antwort wie man C-Programme richtig nach Forth konvertiert. Es reicht nicht einfach nur einen Compiler zu verwenden, sondern man muss den Algorithmus rekursiv definieren.

In der Informatik wird Rekursion unter dem Stichwort L-System und kontextfreie Grammatiken untersucht. Ein Lindenmayer-System ist harmlos es wird nur zur Darstellung von Bildern eingesetzt. Aber was wäre wenn nicht Bilder sondern Algorithmen das Ziel ist, das L-System also ein Programmablaufplan darstellt? Dann haben wir es mit Fraktalen Automaten zu tun, was in Richtung Super-Turing-Maschinen geht. Man hat eine sehr allgmeine Grammatik die ausführbaren Code erzeugt der wiederum ein Problem löst. Die Frage ist jetzt wie muss die Grammatik aussehen um konkrete Aufgaben zu lösen?

Eine konkrete Vorstellung wie sowas praktisch realisiert wird hat [5] Es geht in dem Paper um kontextfreie Grammatiken um Baumartige Algorithmen auszubilden. Ähnlich wie bei den L-Systemen werden hierarchiche Strukturen erzeugt, nur dass sie diesmal ausführbaren Code darstellen. Auf Seite 7 ist eine Abbildung zu sehen die das ganze in Aktion zeigt. Oben ist eine relativ simple Produktionsgrammatik abgedruckt und darunter der Baum der durch diese generiert wird. Ein bisschen erinnert das ganze an den Quatsch-Generator Scigen womit man Texte erstellen kann die so aussehen als wären sie von einem Wissenschaftler erstellt worden. Auf den Einsteiger mögen solche Ansätze wie Voodoo erscheinen, also eine komplett abgedrehte Spielart in der Informatik darstellen, also ein Randthema was sich weit außerhalb des Mainstreams befindet. Doch fragt man einmal Google was es sonst so über “"Tree Based Genetic Programming" weiß, so finden sich fast 1000 Paper zu der Thematik. Und wenn man den Problemkreis etwas erweitert sind noch viele weitere Veröffentlichungen sichtbar. Kurz gesagt, das ganze ist keineswegs ein Randthema, sondern ein umfassend erforschtes Thema was man als langweilig bezeichnen darf.

Die Idee dahinter ist simpel: es geht darum, nicht selber zu Programmieren, sondern stattdessen mathematische Algorithmen zu erfinden die Metaprogrammieren erlauben. Also eine Grammar zu bauen, die Programme erzeugt, die andere Programme überprüfen um so am Ende eine Programmsynthese zu erreichen. Der Software gibt man als Input die Aufgabe wie “bewege den Roboter ins Ziel” und über einen rekursiven tief-verschachtelten stackbased Algorithm wird dann eine Lösung generiert. Ungefähr das ist es was die Autoren anstreben, damit vertrödeln sie ihre Zeit. Ja anders kann man es nicht bezeichnen weil ein praktischer Nachweis der Realisierbarkeit steht

[9] http://www.cis.upenn.edu/∼cis110/13fa/hw/hw08/FloodFill.java

noch aus. Üblicherweise wird die Grammar oder die Rekursiven Algorithmen keiner praktischen Evalution unterzogen beispielsweise in Roboterwettbewerben. Es sind nichts anderes als größenwahnsinnige Konzepte die in einem wissenschaftlichen Paper utopische Vorstellungen vermitteln. Die Botschaft lautet im Kern, dass man das Programmieren selbst an die Maschine verlagern kann, also eine Maschine dazu zu bringen sich selbst zu verbessern.

Gegenmodell Rekursive Algorithmen inkl. der Erweiterung des Genetic Programming sind nichts neues in der Geschichte der Informatik. Sie hatten ihre Blütezeit in den 1960-1970'er Jahren als erstmals die Chaostheorie mathematisch beschrieben wurde und die ersten Computerprogramme entwickelt wurden, welche das ganze auf die Informatik übertragen haben. Im Grunde sind heutige Paper nichts anderes als aufbereitete Informationen die schon damals bekannt waren. Interessanterweise gab es in den 1970'er zu genau dieser Form von künstlicher Intelligenz eine Gegenbewegung. Sie wurde iniitiert von J.C.R. Licklider der sich weniger als Mathematiker wie Donald E. Knuth verstand sondern sich in den geistes- und sozialwissenschaften zu Hause fühlte. Licklider wollte keine kontextfreien Grammatiken erstellen die Computerprogramme synthetisieren sondern Licklider hat die Idee einer Mensch-Maschine-Schnittstelle propagiert. Genauer gesagt hat er das Intergalactic Network, besser bekannt als Internet erfunden.

Während bei rekursiven Grammarbasierenden Suchalgorithmen die Ausgabe von mathematischen Algorithmen und kurzen Codesegmenten im RAM des Computers erzeugt werden besteht bei einem Rechnerverbund der Fokus auf den Nutzereingaben der Menschen. Das Internet zeigt genau das an, was irgendwo auf der Welt in ein Terminal eingegeben wurde. Licklider hat rückblickend die richtige Vision gehabt, seine Idee hat sich durchgesetzt, das rekursive genetic based Programming dagegen nicht. Anders formuliert, es nicht möglich mit einigen simplen mathematischen Formeln eine hohe Komplexität zu erzeugen. Ein Fraktal mag dynamisch aussehen so als ob da verborgene Strukturen enthalten sind die man genauer untersuchen kann doch die darin enthaltene Entropie ist nur scheinbar hoch. Machen wir es konkreter: Angenommen man hat ein hochkomplexes Fraktal vorliegen in 10000x10000 Pixeln, dann kann man dieses komprimieren auf die Erzeuger-Formel, also jenen Algorithmus womit es generiert wurde. Dieser Code war nur 5 kb groß. Anders formuliert, so komplex wie gedacht ist das Farbenspiel nicht, sondern 5 kb ist nicht mehr als eine Strophe von einem Gedicht.

4.5 Genetic Programming

Anfangs glaubt man vielleicht dass rekursive Algorithmen, Chaostheorie und Erzeugergrammatiken etwas sehr mächtiges und wenig erforschtes Thema seien zu dem man unbedingt mehr erfahren müsste weil davon die Zukunft der gesammten Menschheit abhängt, doch wenn man einmal Google Scholar fragt was es über "genetic Programming" schon heute weiß so findet sich erstaunlich viel Material zu diesem Thema. Nicht weniger als 134000 Paper werden angezeigt die allesamt in angesehenen Fachzeitschriften veröffentlicht wurden. Wenn man weiß, dass über alle Wissenschaften hinweg es weltweit nicht mehr als 50 Mio Paper gibt dann ist das eine ganze Menge (0,268% genau gesagt). Selbst über ein sehr bekanntes Thema wie Keuchhusten (whooping cough) gibt es nur halb so viele Paper.

Allein im Jahr 2017 sind bereits 4500 Paper zu "genetic programming" erschienen und das Jahr ist nochnichtmal vorbei. Es wird in sehr vielen Kontexten verwendet und offenbar glauben die Autoren, dass es echte Wissenschaft wäre. Ein bisschen abgedreht aber mathematisch fundiert. Und genau das scheint das Problem damit zu sein, das ganze ist eben keine absonderliche Theorie von Außenseitern sondern offenbar Ausdruck eines Selbstverständnis. Wer sich irgendwie zugehörig fühlt zur Informatik veröffentlicht ein Paper über dieses Thema. Leider ist inhaltlich das ganze nicht beosnders ergiebig. Es wurde bereits alles darüber gesagt, und das sich daraus neue Anwendungen ergeben ist unwahrscheinlich. Die hohe Anzahl der Paper verrät eher wie sich die akademische Informatik selber versteht. Und zwar möchte man sich gegenüber dem stupiden Programmieren abgrenzen. Echte Software mit C oder Java zu schreiben, das ist etwas was die Studenten machen und vielleicht noch Linus Torvalds, wir jedoch die echten Informatiker an den Hochschulen schreiben Codegeneratoren. Also Programme mit denen sich andere Programme erstellen lassen, wir stehen oben auf dem Berg und die anderen sind von uns abhängig, so jedenfalls die Ideologie.

Aber welcher Nachteil entsteht genau wenn man keine Genetic Programming Paper und keine dementsprechende Software besitzt? Dann ist man zurückgeworfen auf seine IDE wo man in Java oder einer anderen Sprache manuell etwas programmiert und sich via github Anregungen holt wie es besser geht. Aber ist das wirklich eine Verschlechterung? Ist nicht vielmehr die manuelle klassische Programmierung das worum es wirklich geht? Ja ich würde sogar noch eine Stufe weiter gehen und "genetic Programming" mit dem Unsinnsgenerator Scigen vergleichen wollen. In beiden Fällen wird die Erstellung von Content an eine Maschine verlagert und in beiden Fällen ist das Ergebnis kompletter Unfug. Es macht zwar Spaß sich damit zu beschäftigen, es ist lustig sein Gegenüber zu veräppeln, aber es bringt die Wissenschaft nicht weiter.

Anders formuliert, Genetic Programming ist keineswegs hardcore Informatik die anstrengend ist, sondern es ist ein Thema mit man sich gerne und zur Entspannung beschäftigt. So ähnlich wie Physiker wenn sie schon leicht angetrunken sind, auch mal näher über ein Perpetum Mobile diskutieren oder sogar behaupten eines gebaut zu haben was sie aber nicht vorführen weil das Patent noch nicht beantragt wurde.

Warum diese Dinge so beliebt sind hat etwas mit den sozialen Rollen zu tun innerhalb derer sich Wissenschaftler befinden. Die Gesellschaft erwartet von einem Physiker im Grunde, dass er ein Perpetum Mobile erfindet. Weil so eine Maschine genau das ist, was alle wollen. Und en Informatiker soll doch bitteschön Metaalgorithmen erfinden die sich selber programmieren. Insofern sind die Paper über dieses Thema der Versuch den Rollenerwartungen gerecht zu werden. Genauer gesagt mit jenen Rollenerwartungen welche Informatiker entdeckt haben.

Als Negativ-Beispiel möchte ich das Paper [1] zitieren. Es ist gerade in diesem Jahr erscheinen und kann als typisch gelten. Es geht darum um Genetic Programmierung auf einer Two-Stack GPU Grafikhardware. In dem Paper enthalten sind Literaturlisten, Algorithmen und Tabellen. Ferner gibt es noch Erläuterungen wie das Projekt durchgeführt wurde. Die Autoren glauben offenbar, dass jemand da draußen diese Anforderungen an die Informatik gestellt hat, und die Autoren liefern jetzt. Aber will die Gesellschaft wirklich derartige Algorithmen haben, bringt das die Dinge voran? Zunächst einmal wäre es sicherlich toll, wenn man damit komplexe leistungsfähige Software generieren könnte. Also schön mit Grafikkarten von nvidia und ein wenig Hokus pokus Software automatisch erzeugen, verifizieren und auf Realen Systemen deployen. Doch leider zeigt die Erfahrung dass es so nicht geht. Das ganze erinnert doch stark an ein Perpetum Mobile wo nicht Erkenntnis sondern Aberglauben das Ziel ist.

5 Hardware

5.1 Traumcomputer GA144

Der GA144 wird manchmal als Superchip bezeichnet. Seine Faszination erklärt sich daraus, dass heute übiche PCs und Microcontroller

rund 2 MIPS je Milliwatt (Million Instructions per seconds) benötigen, der F18 Chip aus dem der GA144 besteht jedoch 155 leistet. Woran das liegt ist simpel: Stackmachine plus Clockcless CPU plus fehlender Microcode = supereffizienter Microchip. Es bleibt nur ein Problem. Wie programmiert man diesen Chip? Einen Einstieg liefert [10] dort wird das übliche Märchen vom schlanken Forth verbreitet, dass es also nur zwei-drei Register gibt einen Instruction Pointer und ganz wenig Forth Kommandos. Nur das ist nicht die volle Wahrheit. So wie in dem Diagramm gezeigt funktioniert der Chip nicht. Eigentlich ist es sehr viel komplizierter. Und zwar muss man zuerst einmal Mono-Develop bei sich installieren, dann noch f-sharp und dann ontop den GA144 Simulator. Als “simple” würde ich das nicht bezeichnen eher als Schwergewicht mit massiver Überladung.

Was in der Anleitung leider nicht thematisiert wurde, und wo ich ebenfalls passen muss ist die Frage wie man auf dem GA144 Chip ein Computerspiel startet wie z.B. OpenRA. Der mitgelieferte RAM Hauptspeicher reicht jedenfalls nicht aus um Windows 10 zu booten. Er beträgt magere 2 MB. Das ganze lässt viele Fragen unbeantwortet, aber vielleicht will ich auch gar nicht wissen wie man OpenRA auf dieser Höllenmaschine spielt? Hoffentlich hat das nichts mit neuronalen Netzen zu tun, das wäre nicht gut.

Ich sage hoffentlich, weil es irgendwie auf der Hand liegt. Neuronale Netze dürften ungefähr das sein, was Chuck Moore als sourceless Programming definiert hat, also Software die man nicht programmiert sondern irgendwie von alleine da ist. Ferner basieren Neuronale Netze auf Rekursion und sie benötigen schnelle Hardware.

Auf der Euroforth 2016 Tagung hat jemand ein rekurrentes Neuronales Netz vorgestellt was in 900 Lines of Code in Forth programmiert wurde. Würde man das auf den GA144 Chip portieren hätte man einen leistungsfähigen Neurocomputer. Da ich mich ein wenig mit neuronalen Netzen beschäftigt habe, kann ich dazu ein wenig mehr erzählen. Die Kurzfassung lautet: es bringt nichts.

Die Langfassung: von den Leistungsdaten ist ein GA144 Chip rund 50x energieeffizienter als ein normaler Chip der in C programmiert wurde. Das heißt, wenn man den Computer mit 10 Watt betreibt ist er 50x schneller als ein normaler Computer. Das neuronale Netz also seine Fähigkeit sich an externe Muster anzupassen funktionert so wie es auch neuronale Netze auf normalen PCs tun. Wir erhalten also eine Neuronalen Netz Simulator der leicht effizienter arbeitet als gewöhnlich. Leider bringt das null Vorteil. Es wird nicht gelingen damit auch nur simple Mathematische Gleichungen wie z.B. Primzahlen zu lernen. Der praktische Nutzen ist gleich Null. Trotzdem rate ich explizit dazu, das einmal auszuprobieren.

Also schön mit Forth ein neuronales Netz auf einem GA144 schreiben, dass dann auf 100% CPU Load hochfahren und dabei zuschauen wie die Lernschritte durchlaufen. Zu Anfang wird das neuronale Netz seinen Fehlerwert senken und man hat den Eindruck, als ob es gut funktioniert, doch irgendwann wird sich der Errorwert einpendeln und dann passiert gar nichts mehr. Unverändert laufen die Cores auf maximum Auslastung ohne dass das neuronale Netz sich weiter verbessert. Das heißt in der Realität dass es nicht in der Lage ist, eine Reihe mit Primzahlen fortsetzen oder bei einer inversen Kinematik die Zielkoordinaten zu bestimmen.

Ja würde man die Geschwindigkeit des Systems nicht nur um den Faktor 50 sondern um den Faktor 50000 erhöhen, würde das neuronale Netz immernoch nicht lernen. Es wäre in einem sogenannten lokalen Minimum gefangen und ist nicht der Lage die Gewichte so zu optimieren, dass die Trainingsdaten verknüpft werden. Das Phänomen mit ist bekannt aus neuronalen Netzen die auf herkömlicher Hardware entwickelt werden, und es tritt auch bei nvidia Grafikkarten, Supercomputer, Forth Chips und sogar bei optischen DSP Chips auf. Anders formuliert, so künstliche Intelligenz erzeugen zu können hat von der Praxis keine Ahnung.

Man kann nicht nur den GA144 verwenden um neuronale Netze zu simulieren, sondern das ganze gleich als FPGA aufbauen, also nicht mit der Forth Programmiersprache dazwischen sondern die Lernalgorithmen direkt im FPGA synthetesieren. Die Effizienz des Chips ist dann noch etwas höher und es gibt mehrere Paper wo das ausführlich beschrieben wird. Nur, auch das ist Pseudoscience, weil die Autoren nicht willens oder nicht fähig sind das ganze als Failed Project zu betrachten. Sie glauben sie müssten das Thema unter didaktischen Punkten jemanden erklären und das würde sie davon befreien die Schwächen bloßzustellen.

Selbst Projekte die halbwegs seriös daherkommen, beispeilsweise wenn man Convolution Neural Networks zur Bilderkennung in einem FPGA einprogrammiert, sind objektiv betrachtet Zeitverschwendung. Würde man die Bildparser ohne Neuronales Netz sondern als Scripting AI in Java programmieren, wäre es um einiges effizienter. Neuronale Netze und hocheffiziente Stackmaschinen sind nichts anderes als eine Modeerscheinung die Konkunktur hat, weil es alle so machen.

Schauen wir uns den GA144 Chip genauer an, genauer gesagt das Entwicklerboard auf dem er sich befindet. Der Hauptkritikpunkt ist der winzige RAM. Auf dem Chip selber ist nur der Stack verbaut, extern auf dem DEveloperboard befinden sich 2 MB Ram. Viel zu wenig für eigene Programme erst recht nicht für ein Betriebssystem. Wollte man einen halbwegs ordentlichen Rechner haben, kann man ruhig einen altersschwachen Intel Atom Prozessor nehmen aber dem dann 1 GB RAM spendieren. Dort lädt man Linux hinein und ontop noch eine Bilderkennungssoftware die 10 MB an Sourcecode mitbringt. Man erhält darüber ein extrem leistungsfähiges System, und das obwohl oder gerade weil die CPU ineffizient ist und die Hardware fast nichts kostet.

Es handelt sich um eine Kuriosität innerhalb der Informatik. Auf der einen Seite werden riesige Forchungsprojekte durchgeführt bei denen neuronale Netze in driverless Cars, und zur Steuerung von OP-Robotern im Krankenhaus eingesetzt werden, auf der anderen Seite fragt niemand ob neuronale Netze überhaupt in der Lage sind, auch nur Primzahlen zu erkennen. Also die Frage “ist 113 eine Primzahl?” korrekt zu beantworten. Neuronale Netze können diese Frage nicht beantworten. Und der Grund warum das niemandem auffällt und die Frage danach bereits an Blasphemie grenzt wirft kein gutes Licht auf das Wissenschaftssystem. Kennzeichen von Protowissenschaft ist es, dass zwischen Magie und Rationalität nicht konsequent unterschieden wird um so die sozialen Rollen stabil zu halten.

Primzahlen Vielleicht an dieser Stelle ein kleiner Exkurs zu Primzahlen. Das hat nichts mehr mit Forth zu tun, aber egal. Anders als üblicherweise in den Online-Foren und in den Vorlesungen hat keinerwegs der Fragesteller der wissen möchte ob man mit Neuronalen Netzen Primzahlen testen kann irgendwas nicht richtig verstanden, sondern es liegt an der Informatik diese Frage zu beantworten. Die Erklärung lautet, dass erstens neuronale Netze keine turing-mächtigen Maschinen sind, also gegenüber manuell programmierten C-Code unterlegen sind. Das war auch damals schon beim xor Problem der Fall, und convolutional Neural Networks haben lediglich die Grenzen des möglichen leicht verbessert. Und zweitens, selbst wenn neuronale Netze all das könnten was auch Sourcecode in C kann, würde es dann noch immer nicht funktionieren. Weil, auch beim genetic Programmierung wo eine Turing-mächtige ’Maschine evolviert wird, ist es nicht möglich Primzahltest-Programme zu erzeugen. Rein theoretisch schon, nur leider würden ca. 1 Mio Jahre vergehen bis man das passende Program im Gödelraum gefunden hat.

[10] https://blogs.msdn.microsoft.com/ashleyf/2013/10/13/programming-the-f18/

5.2 Hardware-Rezension

Der leichteste Einstieg in Forth besteht darin, es als Programmiersprache zu betrachten. Bei Repl.it gibt es einen Online-Interpreter, man kann aber auch gforth nutzen um kleinere Forth Programme laufen zu lassen. Für den C-64 gibt es eine Forth Distribution namens Durexforth. Allen Forth Implementierungen gemeinsam ist, dass sie Forth-Wörter auf Machinencode mappen. Anders als bei einem C-Compiler wird der Code nur interpretiert, was dazu führt, dass die Geschwindigkeit ähnlich langsam ist als wenn man Python verwendet. Das heißt, Forth ist mindestens 3x langsamer als wenn man C verwendet.

Will man die maximale Performance haben, muss man einen Forth Microcontroller verwenden. Hier wird es schon etwas schwieriger, weil es soviele Forth Chips nicht gibt. In den 1980'er war die Novix Baureihe beliebt, die aber außerhalb der Forth Community niemand verwendet hat. Eine aktuelle CPU ist die GA144 welche zusammen mit einem Entwicklerboard 450 US$ kostet. Laut der Anleitung ist darauf auch ein Steckplatz für SRAM Chips enthalten, so dass man den Hauptspeicher erweitern kann.[11] Ob das so stimmt und wie man das Board n der Praxis einsetzt weiß ich leider nicht. Es gibt nur wenige Blogpostings von Leuten die es verwenden und auch Youtube Videos sind selten. Das Problem sind wohl weniger die 450 US$ für das Evalulation Board inkl. Forth Chip, sondern das Problem dürfte sein, dass die Plattform komplett anders ist als normale Microcontroller. Eines ist jedenfalls sicher, mit dem GCC oder mit LLVM kann man dafür keinen Maschinencode erzeugen. Jedenfalls nicht out-of-the-box.

Rein von den Leistungsdaten sind Forth Chips wohl unglaublich schnell. Über die Novix Chips in den 1980'er wurde hinter vorgehaltener Hand erzählt, dass sie schneller seien als jede andere Microcontroller und das selbe wird über den GA144 Prozessor gesagt. Laut den Leistungsdaten ist die Maschine extrem flott. Das Problem ist auch hier wieder weniger die nackte Leistung als vielmehr die Frage ob man damit klarkommt. Weil, selbstverständlich muss man die Forth Programmiersprache beherschen sonst wird es nicht. Zu Forth gibt es leider keine guten Einführungsbücher und auf Deutsch schonmal gar nicht. Die Lage ist am ehesten vergleichbar mit Ende der 1970'er Jahre als der neue Altair PC gerade vorgestellt wurde, und eine Handvoll Bastler damit herumgespielt hat, lange bevor es die ersten Firmen gab die es auf eine professionelle Ebene gehoben haben.

5.3 Arduino an der Fernbedienung

Die einfachste Methode um Roboter jeder Art zu steuern besteht darin, dass man an einen Arduino Kleinstcomputer eine analoge Fernbedienung anschließt. Das ist alles. Damit lassen sich sowohl remote Cars, remote controlled micromäuse oder UAVs steuern. Das Prinzip entspricht ungefähr dem was als Minimal-Forth System bekannt. Dort wurde leicht im Scherz vorgeschlagen dass man nur zwei Befehle implementiert und das System aus der Ferne steuert. Genau dieses Konzept funktioniert ausgezeichnet. Es bedeutet, dass man sich die Bastelleien mit dietlibc Biblitotheken, GCC für embedded, echtzeitfähige Microkernels und Assemblerprogrammierung komplett sparen kann,sondern Elektronik komplett analog ausführt. Das heißt sie so nutzt wie es in den 1950'er Jahren üblich war. Das heißt, die Micromouse besteht aus einem Analogen Schaltkreis und der wird mit dem Arduino gekoppelt. Vom Arduiono aus geht es dann weiter zu einer Workstation wo High-Level-Programme in C# und Python laufen.

Selbstverständlich reicht die Prozessorleistung eines Arduinos nicht aus, um damit Roboter zu steuern. Ja es läuft nochnichtmal ein Betriebssystem darauf, so langsam ist die Hardware. Aber dafür wurde der Arduino auch nie gebaut. Er soll einfach nur ein WLAN Signal was von einer Workstation stammt an einen anlogen Schaltkreis weiterleiten.

6 Compiler friendly

6.1 Microcontroller

Einen ungewöhnlichen aber brauchbaren Ansatz um Microcontroller miteinander zu vergleichen liefert [12] Dort steht nicht etwa die Leistung der CPU im Mittelpunkt sondern es wird gezielt nach der "Compiler friendly"ness gefragt. Also wie leicht sich mit einem Standard-Compiler wie GCC dafür Code erzeugen lässt. Interessanterweise unterscheiden sich die Microcontroller hier beträchtlich. Bei lobenswerten Systemen wie dem MSP430 ist die Unterstützung mit C-Compilern vorbildlich, während andere Systeme wie die PIC Modellreihe häufig Probleme verursachen. In der Art, dass man entweder einen kommerziellen Compiler dazu kaufen muss, oder man nicht in C sondern in einer Sprache programmiert oder im Extremfall dass es überhaupt keine Compiler gibt und man Forth bemühen muss.

Der Fokus auf die Unterstützung durch C-Compiler ist wichtig, wenn man plant die Geräte tatäschlich irgendwo zu nutzen, also damit einen Roboter zu bauen. Das heißt, nicht der controller an sich steht im Mittelpunkt und wie schön doch die LED Lampen blinken sondern worum es geht ist das Ausführen von Software. Das heißt, der MSP430 wird instrumentalisiert und nur als Teil eines größeren Systems betrachtet. Genauer gesagt eines Hardware-Projektes bei dem der Microcontroller nur eine Komponente unter vielen ist.

Als noch verbreiterter gilt der AVR Microcontroller. Dieser lässt sich sehr leicht in C-Programmieren und die Auswahl an Beispielprogrammen und durchgeführten Projekten ist immens. Man kann daraus ableiten, dass eine gute Unterstützung durch OpenSource C-Compiler das wichtigste Krieterium ist ob sich ein Microcontroller im Massenmarkt bewährt. Also wie beliebt er bei den Kunden ist und wieviel sie damit anfangen können. Zum AVR Controller gibt es sogar eine Computerclub2 Sendung wo das Thema sehr einsteigerfreundlich aufbereitet wird.

Das interessante ist, dass auf der Plattform Microcontroller.net es relativ viel Interesse an Forth gibt. Meist von Leuten die Standard-AVR Controller schon ausprobiert haben, dafür in C auch kleine Hello World Programme geschrieben haben und jetzt Lust haben für neues. Sie wählen bewusst oder unbewusst einen compilerunfrinedly Microcontroller der sich nur in Forth programmieren lässt und schauen dann einfach mal wie weit sie damit kommen.

Interessant ist dann auch die Diskussion in dem Forum zu beobachten. Im Regelfall versuchen die Leute zuerst mit Forth klarzukommen, dann versuchen sie das Forth mehr in Richtung C zu trimmen. Üblicherweise geht das nicht. Der Fehler den sie machen ist es, mit Forth Probleme lösen zu wollen die sonst mit C gelöst werden. Also anwenderorientiert zu denken. So nach dem Motto: mein Linefollowing Algoirthmus soll nicht auf einem AVR Microcontroller in C programmiert werden sondern auf einem stackbasierenden Microcontroller in Forth. Wie mache ich das?

Die Antwort lautet: gar nicht. Um Forth zu verstehen muss man sich Probleme suchen, für die Forth konzipiert wurde, also rekusriv Probleme die einen Stack erfordern. Ein guter Anfang ist es, den Roboter so zu programmieren dass er sich wie ein rekursiver Floodfill Algorithmus verhält. Also schön ein Lsystem aufmalt. Oder noch besser, ein rekursives neuronales Netz zu verwenden um das System komplett autonom zu gestalten. Das ist dann echtes Forth.

[11] Laut der Anleitung http://www.greenarraychips.com/home/documents/greg/DB003-110926-EVB001.pdf Seite 6 kann man einen 2 MB RAM Baustein und einen 1 MB Flash Baustein direkt an das Eval Board anschließen.

[12] https://www.mikrocontroller.net/articles/Mikrocontroller_Vergleich

Will man derartige Microcontroller streng nach Forth-Style programmieren muss man ein Konzept wählen was als “Neuromorphic Microcontroller” bekannt ist. Ein Verfahren was neuronale Netze, rekursion und Stackbased Memory miteinander kombiniert.

Forth ist die ideale Sprache wenn man neuronale Netze implementieren möchte. Der Grund ist dass Forth auf Stackmachinen extrem effizient ausgeführt wird und das Forth Programme naturgemäß sehr kompakt sind. Die Funktionalität wird nach dem Starten des Programms entfaltet und passt sich adaptiv an die Umgebung an. Man kann sagen, dass C-Programmierer die Lösung zum Ziel in Form von Ablaufplänen und UML Diagrammen erarbeiten während Forth Programmierer ihr superkurzes Neural Network starten und die Machine dann autonom agiert. Die Kombination aus Forth plus Neuronalen Netzen ist ideal wenn man eine magische Vorstelung von Maschinen besitzt wonach man sich nur etwas zu wünschen braucht was dann in Erfüllung geht. Ein forth-basierendes neuronales Netz ist vergleichbar mit einem Perpetuum Mobile: es ist unglaublich einfach aufgebaut, es ist komplett anti-Mainstream und es ist Ausdruck eines Trends hin zu Neureligiösen Newage Bewegungen wo das persönliche Empfinden im Mittelpunkt steht und etablierte Wissenschaft nicht länger im Fokus steht. Der Forth way of live bedeutet, dass man Rationalität ablehnt und stattdesen komplette Anarchie pflegt. Also die Rebellion ins Zentrum stellt und nach Sinn sucht den das rationale Denken nicht erfüllen kann.

AVR Microcontroller Jetzt stellt sich die Frage warum es ausgerechnet Forth und neuronale Netze sein müssen. Zur Erinnerung: man kann Microcontroller auch normal programmieren. Man sucht sich einfach einen c-friendly Compiler wie den AVR heraus, schreibt sein Programm in C und spielt es auf den Chip. Der Weg dorthin ist simpel, Fehler sind unwahrscheinlich. Man kann darüber eine sehr preiswerte Entwicklungspipeline von einer Idee zu einem funktionsfähigen Roboter realisieren. Er basiert auf Standard-Komponenten wie der genannten Hardware plus OpenSource C-Compiler welche umfassend dokumentiert sind und mit zahlreichen Beispielen im Internet.

Die Nutzung von Altenativen dazu genauer gesagt Forth in Kombinination mit Neuronalen Netzen und rekurrentem Memory ist hingegen etwas außergewöhnliches. Von den ökonomischen Kosten her gesehen ist es viel zu teuer um damit in vertretbarer Zeit ein Ziel zu erreichen, man wird also am Ende keinen funktionsfähigen Roboter vorweisen können. Eine Möglichkeit die Kosten zu senken gibt es nicht. Weil die Nichtverfügbarkeit von Forth Tutorials, die Nicht-Verfügbarkeit von C-Compilern für PIC Microcontroller und die nicht-wiederholbarkeit von neuronalen Netz Experimenten ist eine Tatsache. Sie führt dazu, dass die gewohnte Engineering Pipeline ins Leere läuft man sich also mit sich selbst beschäftigt und das eigentliche Ziel aus den Augen verliert. Vermutlich ist das der Grund warum, sich einige Leute für Forth und Neuronale Netze stark machen, weil sie mit dem gewohnten C-basierenden Engineering nichts anfangen können, weil sie nicht gewohnt sind Algorithmen zu entwickeln weil sie keine funktiorenden Roboter anstreben. Überlicherweise enden Forth Projekte mit dysfunktionaler Software, also mit Robotern die nicht im Ziel ankommen, Betriebssysteme die nicht booten und Primzahlprogrammen die nicht funktionieren. Gleiches gilt für rekurrente Neuronale Netze. Insofern ist es sinnvoll diese Anti-Technologie zu kombinieren so bleibt es wenigstens überschaubar und die C-Programmierer können sich mit echten Algorithmen beschäftigen wo sie richtige Probleme lösen.

6.2 Forth vs Rest der Welt

Zwischen Forth und Mainstream gibt es einen gewaltigen Unterschied. Schauen wir uns zunächst ein Buch über den AVR Microcontroller an. Ich brauche hier keinen konkreten Titel zu nennen es gibt mehrere gut geschriebene sogar auf Deutsch. Üblicherweise steht in solchen Büchern das Projekt im Fokus. Meist wollen die Leser wissen wie man einen Roboter baut um beim Micromouse Wettbewerb teilzunehmen. Und die dazu nötigen Kenntnisse werden in dem AVR Buch vermittelt. Es wird der Microcontroller selber beschrieben, es wird erklärt wie man mit GCC dafür ein Programm erstellt, welche Variablen dort enthalten sind, wie man das auf die Hardware draufspielt und den Code verändert. Manchmal gibt es noch einige Basisalgorithmen wie Pfadplanung und Steering so dass man der Lektüre gleich anfangen kann mit seinem eigenen Hobbyprojekt. Es handelt sich also um einen produktiven aufwärtsgerichteten Lernprozess der an einem Ziel ausgerichtet ist und es üblicherweise auch erreicht.

Jetzt schlagen wir ein Buch über Forth auf. Davon gibt es nicht so viele. Im Regelfall ist man dort nicht auf konkrete Projekte fokussiert sondern erklärt erstmal was eine umgekehrte Polnische Notation ist. Dann erfährt man dass man umdenken muss und sich einen Stack vorstellen muss, weil es sonst nicht geht mit der Programmierung. Dann wird noch kräftig über C-Compiler abelästert weil die angeblich ineffizient wären. Eine Auflistung über vorhandene Forth Bibliotheken mit denen man gleich losprogrammieren kann fehlt natürlich, stattdessen wird ausschweifend erklärt, dass sich jeder seine eigene Software from scratch schreibt weil das effizienter wäre und die Hardware besser auslastet. Aha, und irgendwas konkretes Anfangen lässt sich mit dem Wissen auch nicht, es sei denn man fühlt sich berufen Forth weiterzuempfehlen und eigene Paper zu diesem Thema zu schreiben.

Aber wie kommt eigentlich der fundamentale Unterschied zwischen Forth und dem Rest der Welt zustande? Nun, ein AVR Microcontroller und der passende C-Compiler wurden entwickelt um die Dinge voranztubringen also in Projekten konkrete Ergebnisse zu erzielen. Es geht darum, äußere Anforderungen einzuhalten und funktionsfähige Hardware plus Software zu liefern. Es sind Systeme für den Praxiseinsatz und dementsprechend ist auch das Entwicklungsmodell. Bei Forth hingegen geht es um das genaue Gegenteil. Es geht bei Forth niemals darum Roboter zu bauen, Software auszuliefern oder das Programmieren zu lernen sondern es geht um Forth an sich. Es handelt sich um eine negative Lernkurve wo man sich all jene Fähigkeiten abgewöhnt auf die es ankommt um Erfolg zu haben. Es ist nicht übertrieben C-Compiler für Microcontroller als gut und Forth als böse zu bezeichnen. Das eine strebt einen aktiven Umgang mit Computern an, das andere fördert Aberglauben und Desinformation.

6.3 Ein C to Brainfuck Compiler

Was Forth ist wurde bereits angedeutet, ein Problem was einer Lösung bedarf. Forth ist die Frage, ein C-Compiler die Antwort. Genauer gesagt geht es darum, wie man C welche die wichtigste Programmiersprache ist, auf Architekturen auszuführen die als c-unfriendly bekannt sind. Dabei ist Brainfuck ein idealer Kandidat dafür. Es gilt als noch unfreundlicher zu C als Forth, es enthält nochnichtmal einen Stack. Und oh Wunder, es gibt tatsächlich einen C to Brainfuck Compiler [13] Dieser unterteilt das Band zunächst einmal in Bereiche wie Stack, Heap, und Marker. Um darauf dann Code auszuführen.

Es gibt noch weitere Projekte dieser Art, als Motivation geben die Autoren überlicherweise “spaß an der Sache” an, also das Beschäftigen mit sinnlosen Dingen zum Zeitvertreib. Eines dieser C to brainfuck Projekt lautet treffenderweise Fuckbrainfuck, was wohl andeutet, dass

[13] https://github.com/benjojo/c2bf

man sich von der kryptischen Brainfuck Sprache nicht einschüchtern lässt und sie im Griff hat aber nicht umgekehrt. Mit “im Griff haben” ist gemeint, dass man nachdem der Compiler auf das Ziel einmal eingestellt ist, man dann Pacman in Brainfuck spielen kann und im Extremfall sogar ein Linux darauf laufen lassen kann. Also die Sprache Brainfuck instrumentialisiert für konkrete Anwendungen.

Anders ausgedrückt ist die Gemeinsamkeit von Brainfuck und Forth dass sie auf langjährige C-Programmierer provokativ wirkt und man sich herausgefordert fühlt. Ein C-Compiler für diese esoterische Plattform ist der Beweis, dass man der Herausforderung gewachsen ist. Das man es also fertig bringt auch auf sehr abseitigen Systemen C laufen zu lassen.

6.4 Esoterische Programmiersprachen

Unter der URL Esolang [14] findet sich eine Auflistung von esoterischen Programmiersprachen. Der Begriff ist relativ jung und bezeichnet Sprachen die anders sind als C. Aufgeführt werden neben Brainfuck auch Game of Life auch Intercal und Forth. Forth wird nicht direkt als esoterisch bezeichnet kann aber mit dieser Kategorie aufgezählt werden. Die Gemeinsamkeit von Forth und Brainfuck ist, dass es für beide Sprache Bedarf nach einem C-Compiler in diese Sprache gibt. Wobei der Bedarf bei Forth noch höher ist, gewissermaßen befindet sich Forth in einer Schonzone weil viele behaupten es wäre eine normale Programmiersprache während C esoterisch sei. Aber diese Ansicht ist standard, auch Programmierer die sich mit Brainfuck vertraut machen glauben danach dass die Welt außerhalb irgendwie merkwürdig sei. Dementsprechend als Sakrileg wird es empfunden wenn Compiler existieren die in diese Sprache übersetzen können weil das deutlich macht welche Sprache mächtiger ist.

Das interessante an den Esoterischen Programmiersprachen ist, dass es sehr einfach ist, sich eine neue auszudenken und dafür einen Interpreter zu schreiben, es aber ungleich komplexer ist einen ein vorhandenes C-Programm in diese Sprache zu übersetzen. Auch der C to Brainfuck Compiler gilt zurecht als Meisterleistung, nicht jeder ist dieser Aufgabe gewachsen. Es bedeutet buchstäblich, die Kontrolle über die esoterische Sprache zu erlangen, also deutlich zu machen wer die Deutungshoheit besitzt und wer nicht.

Um die Bedeutung etwas zu erläutern kann man sich Game of life anschauen, eine Programmiersprache die ebenfalls nicht kompatibel ist zu C. Viele Jahre galt die Sprache als uneinnehmbare Festung, es war bekannt wer sie erfunden hat aber es war unklar wie ein C-to-Gameoflive Compiler ausseehen muss. In 2002 wurde unter der Bezeichnung “Life Universal Computer” ein derartiger Compiler vorgestellt. Nicht direkt von C ausgehend sondern von einer vereinfachten Sprachsyntax aber immerhin. Gewissermaßen ist das die erste Schneise in das Dickicht. Mit weiteren Anstrenungen kann man darauf aufbauend einen echten C-to-game-of-live Konverter schreiben um dann Linux auf einer Game-of-Life Installation laufen zu lassen. Wiedereinmal geht es darum, sich nicht von Esoterischen Programmiersprachen einschüchtern zu lassen sondern die C-Weltsicht auf alles zu übertragen was halbwegs nach Computer aussieht.

Was bedeutet es genau eine UTM für Game-of-Live zu schreiben.? Eigentlich dasselbe als wenn man einen C-to-brainfuck Compiler baut. Man unterteilt das Band erstmal so wie es ein C-Compiler gerne hat, also schön mit Stack, Heap und Markern um dann anschließend beliebige C-Programme darauf lassen zu lassen. Es gibt auf Youtube ein Video wo man sieht wie in Game-of-Live der Stack anwächst. Das bedeutet, dass man ein wichtiges Bauelement geschaffen hat. Genauer gesagt dient eni kontrollierbarerer Stack dazu den C-Compiler in Betrieb zu nehmen. Also am Ende ein normales C-Programm auf diesem Stack laufen zu lassen. Oder es etwas umgangssprachlicher zu formulieren: sich über Game of Live zu erheben und es als Esoterische Programmiersprache zu definieren die man selbstverständlich programmieren kann in C.

Game of Life to UTM Schauen wir uns etwas genauer an, was laut dem Esolang WIki im Jahr 2010 entwickelte wurde. Im Grunde wurde ein altes Rätsel geknackt und zwar die Frage wie man einen Compiler entwickelt der ein C-Programm für den Game-of-life Automaten bereitstellt. Sehr bescheiden hat der Autor dieses Programms dann nur den absolut nötigsten Teil veröffentlicht weil er wusste dass es ausreicht. Genauer gesagt wurde nicht der komplette C-to-GameofLife compiler publiziert, sondern nur der Teil um eine UTM Notation nach Gameoflife zu transformieren. UTM selber ist ebenfalls eine Esoterische Programmiersprache aber eine die etwas verbreiteter ist als Game of live. Will man ein C-Programm in einer Game-of-life Simulation ausführen muss man folgende Pipeline anwenden:

C Sourcecode -> Brainfuck -> UTM -> Game of life

Das ist schon alles. Für alle Schritte in der Pipeline gibt es Software. Im Schritt 1 wird das C Programm für Brainfuck aufbereitet, das heißt auf dem Band wird exakt dasselbe ausgeführt was auch das C Programm macht, und das wird dann als Input für den nächsten Compiler verwendet und die UTM Machine wird dann am Ende in Game-of-life als Stack und Heap simuliert.

Anders formuliert eine esoterische Programmiersprache wie Game of life ist keine Gegenposition zur Programmiersprache C sondern es ist ein ausgedachtes Rätsel. Die Intelliektuelle Herausforderung besteht darin einen C-to-Gameof-life Compiler zu schreiben. Also diese abseitige Programmiersprache zugänglich zu machen, sie also in den Griff zu bekommen.

Definition von Forth Mit diesem Hintergrundwissen über den Game-of-Life Automaten lässt sich eine Definition aufstellen was Forth eigentlich ist. Forth ist eine höhere esoterische Programmiersprache. Genauer gesagt ist es ein Rätsel wie Brainfuck bei dem die Lösung darin besteht einen C-to-Forth Compiler zu entwickeln. So bekommt man die Sprache in den Griff.

Selbstverständlich kann man auch Desinformation betreiben und behaupten, dass Forth die echte Programmiersprache wäre und alles andere nur Fake sei. Logischerweise kann man in Forth einen Interpreter für C schreiben oder versuchen Forth Programme nach C zu konvertieren. Nur, diese Desinformation kann man für die anderen esoterischen Programmiersprachen genauso einsetzen. Ja es ist sogar möglich, die Brainfuck Sprache in Hardware nachzubauen.

Anders formuliert, ich glaube das Chuck Moore so eine Art von Scherzbold war, der eine esoterische Programmiersprache erfunden hat und darauf gewartet hat, dass jemand einen C Compiler dafür entwickelt. Und weil das nicht passierte hat er es immer weiter getrieben und CPUs für Forth gebaut die ohne C-Compiler komplett nutzlos sind, sondern wiederum als Rätsel gedacht sind, damit sich die C-Community mit einem Compiler mehr anstrengt.

Nochmal Game of Life Schauen wir uns mal an, was passierr ist nachdem eine UTM Machine für Game of Life veröffentlicht wurde. Im Grunde hat sich dadurch alles verändert. Es macht keinen Sinn mehr direkt in der Game-of-life Syntax sich irgendwelche Patterns auszudenken also Muster die die Bilder auf den Screen malen oder ähnliches. Bis man sich solche Patterns überlegt hat, vergehen Wochen. Stattdessen kann man einfach das gewünschte Muster in der UTM Syntax vorgeben und darüber dann sehr ökonomisch Programme ausführen. Und zwar beliebige Programme die sich leicht erstellen lassen und im Extremfall sogar simpler C-Code sind.

[14] https://esolangs.org/

Ungefähr das gleiche passiert auch mit Forth. Wenn es irgendwann gute Compiler für diese Sprache gibt, wird es sinnlos direkt in Forth zu programmieren. Minimal Instruction Set Computer sind keine Antwort sondern sie sind das Rätsel. Die Herausforderung lautet auf diesen Systemen C-Code laufen zu lassen.

6.5 Forth ist eine Toy-language

Als Toy Language versteht Wikipedia Programmiersprachen die sich nicht für den Produktiveinsatz eignen. Sie sind vergleichbar mit Domain-specific languages und esoterischen Programmiersprachen. Forth erfüllt exakt diese Defintion. Es ist zunächst einmal eine esoterische Programmiersprache (es gibt nur wenig Code in Forth) mit der sich dann demoan-specific Languages erstellen lassen. DSL wiederum sind ebenfalls als Toylanguage zu klassifizieren weil sich damit keine echten Probleme lassen.

Als weitere Eigenschaft von Toylanguages bezeichnet Wikipedia das Fehlen eines Bootstrapping Prozesses, das man also nicht einfach so einen C-Compiler dorthin portieren kann um dann ein Betriebssystem zu programmieren sondern dass die Eigenschaften der Sprache diesen Ansatz erschweren oder sogar unmöglich machen.

Das genaue Gegenteil einer Toylanguage ist C. C wird verwendet um unbekannte / esoterische Architekturen mit Software auszustatten. Mit C werden Programme geschrieben die dann sogar in der Brainfuck Sprache ausgeführt werden. Insofern ist es logisch Forth ebenfalls als Problem zu bezeichnen.

Toylangues zeichnen sich dadurch aus dass wesentliche Elemente von C fehlen: Also Arrays, Records und Pointer. all diese Kriterien erfüllt Forth, es ist eine Sprache der etwas fehlt.

Manchmal wird versucht den Begriff Toylanguage in Frage zu stellen. Das Argument lautet, dass alle Programmersprachen gleichberechtigt währen und Forth wie auch Game of live eine ganz normale Programmiersprache seien und jeder der was anderes sagt das beweisen müsste. Kein Problem hier ist er: Eine Turing-mächtige Maschine ist in der Lage ein Programm auszuführen. Unbestritten sind Game of Life und Forth turing-mächtig. Die Turing-Machine ist jedoch nur die Grundlage für die man Programme schreiben muss um damit etwas sinnvolles anzufangen. Unterlässt man das Programmieren kann man allenfalls Zufallsprogramme ausführen. Ob eine Programmiersprache für die Praxis taugt, also nicht nur eine Toylanguage ist hat damit zu tun, wieviel Geld es kostet dafür ein Programm zu erstellen. Bei Hochsprachen wie C ist es relativ preiswert, deshalb gibt es auch so viele C-Libraries. Bei anderen Sprachen wie Forth ist es hingegen teurer. Der Begriff Toylanguage ist also keiner aus der theoretischen Informatik sondern ist gesellschaftlich definiert. Also welche Funktion Computer und dazugehörige Software haben und wie dessen Weiterentwicklung funktioniert.

6.6 Soziale Implikationen von Stackmachinen

Es gibt eine Reihe von Projekten bei denen Stackmachinen im Mittelmpunkt stehen. Zu nenenn ist einmal die Forth Community rund um Chuck Moore, aber auch einige Dozenten an den Universitäten stellen in ihren Vorlesungen dezidierte Stackcomputer vor. Im letzten Paper von Deepmind wurde eine neural Stackmachine implementiert und die Programmiersprache Factor wird ebenfalls häufiger erwähnt. Wie soll man derlei Projekte einschätzen? Zunächst einmal auf einer sozialen Ebene. Ohne etwas wesentliches zu verpassen kann man in derlei Vorlesungen anfangen mit seinem Nachbarn ein Gespräch über das Wetter anfangen. Und das Deepmind paper über die Neural Stackmachine kann man ebenfalls in den Reißwolf schieben.

Üblicherweise machen die Vertreter von Stackcomputern folgenden Fehler. Sie erfinden zwar eine Rechnerarchitektur mitsamt Machinen Instruction vergessen jedoch einen passenden C-Compiler beizulegen. Das Factor Projekt zeichnet sich dadurch aus, dass eben kein C-to-factor Compiler vorhanden ist, gleiches gilt für Forth CPUs und für die Deepmind Neural Stackmachine. Wenn es jedoch keine C-Compiler in diese Architektur gibt bedeutet es, dass es sich um ein Toylanguage handelt. Man kann das entweder als wertlos bezeichnen oder wenn man etwas mehr Ehrgeiz entwickelt der Gegenseite einmal zeigen was richtige Informatik ist. Dazu muss man einen C-Compiler in diese obskure Sprache programmiren. Also einen C-to-Neuralstackmachine, so dass man die esoterische Architektur ganz normal in C programmieren kann.

Warum es für die Deepmind Neural TuringMachine keinen C Compiler gibt ist simpel. Es ist einfacher sich ein Problem auszudenken als es zu lösen. Und man selber muss das nicht unbedingt tun, weil dann garantiert der nächste ankommt der ein leicht andere Stackmachine vorstellt wiederum ohne C-Compiler dafür. Und dann wiederholt sich das Drama: ohne C-Compiler kann man kein Betriebssystem auf die Plattform portieren und ohne Linux kann man kein Pacman spielen.

Das Problem sind nicht die Stackmachinen sondern das Problem ist, dass ist das soziale Machtgefälle. Das also die eine Seite es lustig findet sich künstliche Probleme auszudenken und dann in diesen Sinnsysteme andere Leute dahingehend zu testen ob sie die Fragen richtig beantworten können. Ginge es in den Vorlesungen an den Universitäten wirklich ums Programmieren und CPUs dann würden als Modellcomputer richtige CPUs behandelt werden für die es bereits C-Compiler gibt. Wo man also standardmäßig ein Betriebssystem und unendlich viel Software drauf laufen lassen kann. Es ist sinnlos sich Problem auszudenken wo keine sind. Man kann sich hunderte von Programmiersprachen und tauusende von Rechnerarchitekturen überlegen die compiler unfriendly sind. Das ganze ist Schwarze Pädagogik in Reinstform. Es dient dazu eine Negativspirale aufzubauen.

Eine Neural Stack Machine [7] zu verstehen ist simpel wenn man den Kontext kennt in dem sie entstanden ist. Wichtigstes Merkmal des ganzen ist es, dass sich die Firma Deepmind einen Scherz erlaubt und eine esoterische Programmiersprache überlegt hat, für die es aktuell noch keine C-Compiler gibt um darüber ein Sinnsystem zu konstruieren indem sie die einzigen sind die wissen wie es funktioniert. Das Paper ist weniger die Antwort auf ein Problem der Informatik, sondern ist ähnlich wie Brainfuck oder Game of Life ein Programmierrätsel. Behauptet wird, dass es sich um ein neuartiges Rechenmodell handelt in Wahrheit fehlt einfach nur ein C-Compiler das ist alles. Die Maschine ist genauso turing-mächtig wie alle anderen Computer, nur ist es eben anstrengend für eine Stackmachine zu programmieren. Den Fehler den Deepmind macht ist, dass sie nicht zugeben, eine Toylanguage erfunden zu haben und das es ein Rätsel sein soll, sondern sie behaupten es wäre echte Informatik. In Wahrheit ist die Neuronale Stackmachine kompletter Unsinn. Man kann mit einem normalen Rasberry PI die selben Berechnungen sehr viel preiswerter ausführen.

Es handelt sich um eine sehr hinterlistige Form von Unsinn weil es für sich betrachtet zunächst Logisch klingt. So als ob da jemand wirklich den Dingen auf den Grund geht und andere einlädt ihm dabei zu folgen. Tatsächlich ist das ganze kein Informatik Projekt als vielmehr ein Pädagogik Projekt. Wer es sich leisten kann, eine Stackmachine vorzustellen und andere in die Rolle drängt das verstehen zu müssen der besitzt soziale Macht. Er kann entscheiden was an den Hochschulen gelehrt wird.

Die Alternative zu diesem Quatschpaper von Deepmind besteht darin, zur Abwechslung mal eine “c-compiler friendly” Architektur vorzustellen. Also einen Computer wofür es bereits einen C-Compiler gibt, auf dem ein Standardbetriebssystem läuft und wo man nichts verstehen muss sondern weitere Programme dafür schreibt.

Angenommen es geht wirklich darum, die Informatik voranzubringen. Welchen Sinn macht da ein Computer auf dem ein C-Compiler

nicht läuft, bzw. erst noch mühsam einer programmiert werden muss? Und wenn es nicht gelingt, hat man ein System ohne Software, also genau das, was nicht das Ziel ist. Es sei denn man will die Informatik nciht voranbringen sondern empfindet sie als Bedrohung. Dann hat man ein Interesse an Inkompatibilität, dann ist man stolz darauf dass C-Programme nicht ausführbar sind.

6.7 C Compiler

Nachdem erläutert wurde, warum Forth, Brainfuck und Game of Life ungeeignet sind soll einmal die Programmiersprache C etwas genauer untersucht werden. Gibt es irgendwas was falsch ist mit C? Wohl kaum, C ist die wichtigste Programmiersprache. Wenn man sie auf einer c-friendly Hardware ausführt erhält man eine Programmierumgebung mit der man weitere Dinge machen kann. Aber warum hat C dann so schlechtes Image? Warum wird an den Universitäten alles mögliche erzählt nur nicht das C super ist?

Schauen wir uns dochnochmal die Welt außerhalb von C an. Nehmen wir mal an, man sich eine Stackmachine überlegt auf der man Forth zum Laufen bringt. Im Grunde hat man alles getan um ein Bootstrapping zu verhindern. Genauer gesagt hat man sich den Ast abgesägt auf dem man sitzt. Üblicherweise sind Forth Programmierer nicht in der Lage in Forth einen C-Compiler zu programmieren der auf ihrem System läuft um damit weiteren Code zu übersetzen der schon da ist. Und das bedeutet nichts anderes, als dass die neuentwickelte Plattform ihnen ganz allein gehört. Sie können sich Aufgaben überlegen von denen die Antwort bekannt ist und die unter C keinen Sinn machen. Ein Primzahlprogramm in C zu schreiben ist simpel, über Copy&Paste hat man das in weniger als 10 Minuten erledigt. Bei Forth hingegen ist es schwerer. Erstens läuft der C-Code nicht und will man es in Forth nachprogrammieren dauert es.

Häufig wird argumentiert dass eine Stackmachine mitsamt Forth das Denken in ungewohnten Bahnen trainieren würde. Das Gegenteil ist der Fall. Es ist nichts anderes als Latainunterricht der mit einer Prügelstrafe angereichert wurde. Nicht um die Schüler geht es, sondern um die Bedürfnisse des Lehrkörpers. Forth ist eine sehr effektive Methode wie man Informatikstudenten zu Versagern erklären kann. Es ist ein Sinnsystem was sie nicht durchschauen und wo der Lehrer der einzige ist, der zwischen RIchtig und Falsch unterscheidet. Forth ist eine autoritäre Unterrichtsmethode welche über Strafe funktioniert. Nicht Erkenntnis ist das Ziel sondern Gehorsam. Forth hat die Aufgabe, eine Gegenöffentlichkeit zu bauen, wo man mit seinen C-Kenntnissen nichts anfangen kann. Forth kann man nicht verstehen oder lernen sondern Forth kann man nur hassen. Dieses Gefühl ist normal und es ist so gewollt. Es soll einschüchtern, gefügig machen und Abhängigkeit herstellen.

Die einzig richtig Antwort auf Forth besteht darin, sich darüber lustig zu machen. Forth mag es nicht, wenn man es also Toylanguage bezeichnet, Forth hat ein Problem infrage gestellt zu werden. Forth will nicht verspottet werden.

Jokes über Forth finden sich hier [15]

6.8 C-friendly Compiler

Um auf die Besonderheiten von Forth einzugehen ist es wichtig zunächst einmal sprachlich zu beschreiben, woran die Welt außerhalb der Forth Community interessiert ist. Neue CPU die auf den Markt kommen werden nach einer Maßzahl namens “c-friendly” eingestuft. Es handelt sich dabei um Bedürfnisse von G-Compilern wie GCC. Üblicherweise benötigen solche Compiler CISC CPUs mit vielen Registern. Dafür können sie guten Maschinencode erzeugen. Wenn eine CPU diese Kriterien nicht erfüllt wie z.B. die PIC CPU und erst recht die Stackmaschinen von Chuck Moore, gelten diese CPU als c-unfreundlich und werden als schwer zu programmieren bezeichnet. Es gelingt nicht oder nur unter großem Aufwand dorthin einen C-Compiler zu portieren. Und ohne C-Compiler kann man auch keine Betriebssysteme auf diese Plattformen bringen. Aus Sicht der Ingineure muss man dann über Alternativen nachdenken. Vielleicht etwas konkreter:

Ein Ingineur erhält bei sich auf dem Schreibtisch die neue GA144 CPU. Er schaut in das Datenblatt und erkennt, dass diese CPU nicht C-friendly ist. Sie hat keine Register und noch einige weitere Besonderheiten wodurch es nicht möglich ist, dorthin einen C-Compiler zu portieren. Wenn er dennoch für diese CPU Software entwickeln möchte oder muss dann sind Alternative Ansätze nötig. Und die sind teuer. Konkret geht es darum, für die GA144 Software zu entwickeln wenn kein C-Compiler zur Verfügung steht. Der Ingineur könnte jetzt anfangen, einen eigenen C-Compiler zu programmieren der speziell auf Stackmachinen fokussiert ist, oder er könnte manuell Forth Code programmieren. Beides ist grundsätzlich möglich aber nicht so einfach als wäre es ein c-freundlicher Prozessor.

Die Hersteller neuer CPUs haben üblicherweise ein Interesse an hohen Absatzzahlen. Also entwickeln sie den Instruction Set so wie es angenehm ist für C-Compiler. Dies wird als x86 Standard bezeichnet. Kommen wir jetzt zu den Forth CPUs. Diese sind nicht nach dieser Maßgabe entwickelt worden. Es gibt zwar die CPU, doch fehlt dazu noch die passende Software. Die wichtigste Eigenschaft von Stackmachinen ist, dass sie Probleme erzeugen. Das heißt, sie sind dsa genaue Gegenteil von c-friendly, eben wie ein Fixed Gear Bike.

Wie man für Stackmachinen Software entwickelt ist unklar. Mit einem C-Compiler jedenfalls nicht. Üblicherweise werden Stackmachinen in Forth programmiert und zwar manuell. Das heißt, man schreibt das Programm from scratch und zwar in Standard-Forth. Weil das natürlich seine Zeit dauert, gibt es nur wenige Forth Programme. Dadurch, dass Stackmachinen c-unfreundlich sind, hat sich um Forth eine Aura des elitären gebildet. Es ist kein allerweltsprozessor, manche sagen dass Stackmachinen des Teufels sind. Also böse.

7 Minimalismus

7.1 Was ist Minimalismus?

Die Forth Community gibt sich der Illusion hin, dass ihre Programmiersprache minimalistich wäre. Grund für die Behauptung ist der Codeumfang eines Forth Systems was mit 4 kb im Vergleich zu C sehr kompakt ist. Auch ein Forth Programm was ausgeführt wird ist selten größer als 2 kb. Nicht viel anders versuchen auch die C-Programmierer im Ideal des Minimalismus zu sonnen. Da wird gesagt, dass man mit Hilfe der ulibc sehr kompakte Binarys schreiben kann, nicht ganz so klein wie in Forth aber mit 10 kb immernoch sehr kurz. Auch die C-Programme welche auf Micromouse Microcontroller geflasht werden orientieren sich an einem ästethischen Minimalismus in der Art dass sie auf maximale Effizienz getrimmt wurden. Aber ist Forth und C schon echter Minimalismus?

Im Kern wird als Minimalismus ein selbstgewähltes Bezugssystem verstanden. Konkret die Fähigkeit mit einer geringen Codegröße und einem langsamen kleinen Prozessor auszukommen. Das wird gleichgesetzt mit einem erwünschten Verhalten wodurch die Aufgaben überschaubar werden. Man hat ein sehr abgegrenztes Gebiet auf dem man die optimale Lösung entwickelt und dafür dann eine Belohnung erhält – so die Vorstellung. Leider sind sowohl Forth als auch C-Programmnierer einem Trugbild aufgesessen und zwar der Vorstellung dass ein System überschaubar bleibt, wenn der Hauptspeicher weniger als 100 kb beträgt. Minimalismus ist nichts schlechtes, sondern es kommt auf die Kriterien an ihn zu messen.

[15] http://www.yuksrus.com/computers_programmer.html

ich sage, dass Minmalismus darin besteht, einen Rasberry PI Computer zu nutzen der über 1 GB RAM und 64 GB Flash-Memory verfügt und auf diesem ein Robot-Control-System auszuführen was aus 1 Million Lines of Code besteht. Aus Sicht der heutigen Microcontroller-Community mit ihrer Bare-Metal-Ideologie stellt das ein schlimmes Sakrileg dar, weil nach ihrem Denksystem es sich dabei um aufgeblähte Hardware und Software handelt. Doch im Grunde sind Systeme mit 1 GB Ram die Inkarnation von Minimalismus, nur die Methoden um Komplexität zu reduzieren sind andere. Will man Programme schreiben die aus 1 Million Codezielen bestehen benötigt man selbstverständlich Methoden um die Komplexität zu senken. Wenn man diese nicht besitzt entsteht allzuschnell Chaos und es entsteht Bloatware. Eine Möglichkeit besteht darin, wenn man objektorientierte Programmiersprachen nutzt, eine weitere sind UML Diagramme. Dort gibt es ein Designelement namens Packages, was die Aufgabe hat, viele Einzel-Klassen zu einer höheren Gesamtheit zu vereinen. Wenn man sich das entsprechende UML Diagramm ansieht ist es sehr minimalistisch. Man hat 3-4 Packages die genau bündig auf dem Bildschirm angeordnet sind, einige wenige Verbindungsfeile und das entspricht dann 1 Mio Codezeilen.

Es ist also keineswegs so, dass große Programme und umfangreiche Betriebssysteme automatisch überladen sind sondern worauf es ankommt ist Komplexität zu managen. Wenn man lediglich versucht den Speicherbedarf von Software zu minimieren oder ein C-Programm in 32 kb Flash unterzubringen verwendet man den falschen Ansatz zur Komplexitätsredukution. Es erinnert ein wenig an Cargo-Cult-Science auf diese Weise Probleme lösen zu wollen.

Der wichtigste Maßstab um Minimalismus zu erkennen ist ökonomischer Art. Genauer gesagt die Kosten um ein Projekt durchzuführen. Einmal sind es die Hardwarekosten selber, aber auch der Zeitaufwand bis das Ergebnis erzielt wurde. Wenn es gelingt die Projektkosten zu senken hat man verstanden was Minimalismus bedeutet. Projekte in denen Micocontroller mit C oder noch extremer mit Forth programmiert werden haben die Eigenschaft sehr teuer zu werden. Damit ist gemeint, dass da ein Team aus 12 Leuten über Wochen irgendwelche Änderungen vornimmt nur damit am Ende der Roboter exakt gar nichts tut weil noch irgendwelche Bugs nicht gefixt wurden. Zwar gibt es auch in diesen Teams eine Vision davon wohin das Projekt sich entwickeln soll, aber es ist eine die sich nicht erreichen lässt. Es würde zuviel kosten sie umzusetzen. In Folge dessen laufen Anspruch und Realität auseinander. Vom Selbstverständnis haben die Teammitglieder nach 3 Monaten harter Forth-Programmierung so eine Art von Skynet-Betriebssystem für einen Roboter programmiert, defakto jedoch fährt die angeblich so minimalistische Micromouse gar nicht in Ziel.

Codegröße Wie man in Forth oder in Assembler einen Computer progrmmiert ist einfach. Man schreibt die Routinen die man benötigt hin und fertig. Beispielsweise wird die WLAN Karte vom BIOS erkannt und es braucht nur wenige Codezielen in Forth um die Karte online zu bringen und mit einer Basisstation zu verbinden. Dafür braucht man kein eigenes Betriebssystem sondern das geht in 50 Zeilen auch so. Ebenfalls vom Bios unterstützt wird die Grafikkarte. Benötigt man einen Pixel auf dem Bildschirm aktiviert man diesen. Im Extremfall über zwei kurze Zeilen. Was also soll der Unsinn mit Linux und darauf aufbauenden Systemen?

Schauen wir uns den Linux Kernel in seiner aktuellen Version an. Der Sourcecode des letzten Stable Release benötigt gepackt 98 MB auf der Festplatte. Ist das viel oder wenig? Aus Sicht von Forth ist das viel. Es handelt sich um Bloatware die keinen Zweck erfüllt und Ausdruck von Unverständnis gegenüber der Hardware darstellt. Ein wenig realistischer betrachtet sind 98 MB gar nicht mal soviel. Ein gutes Buch über Compilerprogrammierung benötigt als PDF Datei 5 MB auf der Festplatte. Der Kernel benötigt also nicht mehr Platz als 20 Bücher aus einer gut sortierten Oreilly Serie über systemnahe Programmierung. Und wären es nicht 98 MB sondern 300 MB wäre es immernoch wenig. Weil das Thema komplex ist und viele Dinge beachtet werden müssen. Der Grund warum man dazu Bücher benötigt und es als Linux Kernel niederschreibt hat damit zu tun, dass man darüber das Wissen zugänglicher macht. Man braucht also eben nicht selbst wissen wie man im Bios die Grafikkarte aktiviert sondern der Kernel weiß das. Es ist dort notiert als ausführbarer C-Code. Dieses Prinzip Wissen in Büchern aufzuschreiben und als Kernel zu formalisieren dient dazu Komplexität zu senken. Das heißt die 98 MB große Datei mit den Linux Sourcen ist einfacher als das 2 kb große Jonesforth.

Was die Forth Community als erstrebenswert bezeichnet ist der Rückfall in eine mündliche Kultur. Damit ist gemeint, dass das Wissen über Betriebssysteme nicht in Büchern niedergeschrieben steht und auch nicht als C-libraries maschinenlesbar verfügbar ist, sondern es personenzentriert in den Köpfen der Forth Programmierer existiert. Im Extremfall läuft das auf die Idee des Sourceless Programming hinaus, wo es also keinerlei Bücher gibt und auch keinen Sourcecode, sondern man alles im Kopf memoriert und mit wenigen Bootstrap-Anweisungen einen Computer hochfährt. Dieser Ansatz ist keineswegs etwas was es nur in der Computerbranche gibt, sondern man kann auf diese Weise auch Medizin, Jura und Literatur durchführen. Das heißt, Geschichten werden grundsätzlich nur mündlich weitergegeben, Wissen im persönlichen Gespräch vermittelt, es gibt keine externen Bibliotheken wo man sich frei informationen beschaffen kann. Es handelt sich dabei um Wissensmonopole die aufrechterhalten werden. Derjenige der das Wissen besitzt nimmt eine herausragende Stellung in der Gesellschaft ein. Und wird der Medizinmann entführt, weiß der komplette Stamm nicht mehr wie man Krankheiten heilt, weil es nirgendwo niedergeschrieben steht.

Ungefähr das ist die Ideologie hinter Forth. Das man also nichts aufschreibt und keine Bibliotheken anlegt sondern sich stattdesen trifft und darüber sehr limitiert Wissen weitergibt. Die Idee dahinter lautet, darüber die Komplexität zu senken, also die Menge an Wissen zu reduzieren und die Anzahl von Personen zu senken die damit in Berührung kommen. Wie man das bewertet muss jeder selber entscheiden. Fakt ist, dass der Ansatz das Genaue Gegenteil der Aufklärung und der Erfindung des Buchdrucks darstellt. Forth könnte man also als Rückschritt in eine als positiv empfundene Vergangenheit bezeichnen. Es werden gedruckte Informationen abgelehnt, genauso wie formale C-Libraries abgelehnt werden.

7.2 Ferngesteuerte Roboter

In einem Aufsatz über Forth Programmierung war zu lesen, dass das kleinst mögliche Forth System aus genau 2 Befehlen besteht. Der Befehl 1 wartet auf ein SIgnal von außen und Befehl 2 schreibt das in eine Speicherstelle des Computers. Auf den Micromouse Wettbewerb übertragen bedeutet es, dass man den Microcontroller ausbaut und ihn durch eine Fernsteuerung ersetzt. Das kann sogar eine analoge Fernsteuerung sein, die nichts macht als entsprechend der Fernbedienung die Motoren zu aktivieren. Meiner Ansicht eine ausgezeichnete Idee, weil man sich dann den Aufwand mit den MSP430 Microcontrollern, der C Programmierung und der Forth Programmnierung sparen kann. Das heißt, man baut aus dem Roboter die komplette Logik aus und verwandelt es in einen steuerbaren Aktuator.

Gleichzeitig wird damit der Weg frei um ernsthaft über Künstliche Intelligenz nachzugrübeln. Weil nach wie vor die Frage im Raum steht welcher Algorithmus die Fernbedienung steuert. Rein technisch könnte das eine hochgezüchtete Workstation sein, also 16 GB RAM, Fedora Betriebssystem, High-End Grafikkarte. Man verzichtet also nicht wirklich auf den Part mit dem Computer, sondern spaltet das

Problem lediglich auf. In der MIcromouse gibt es einen Analogen Empfänger der tatsächlich nur aus simplen Schaltkreis besteht und auf der Sender-Seite hat man dann einen vollausgestatteten Computer mit allen Extras. Was man nicht mehr hat ist Microcontrollerprogrammierung. Also keine in Assembler geschriebenen Programme, auch keine Forth Scripte mehr, sondern stattdessen hat man Java Programme die viele Megabyte benötigen wenn man sie starten will.

Technisch ist das machbar nur verletzt es die ungeschriebenen Regeln. Unter einem Micromouse Roboter wird derzeit zwingend ein Microcontroller verstanden, also 16-bit Kleincomputer mit 16 kb RAM der in C programmiert wird und wo man das Programm auf den Chip draufflasht. Es geht also um autonome Roboter, die ihre Computer onboard mitführen. Aus Sicht der Künstlichen Intelligenz ist das jedoch das selbe, als wenn man die Micromouse mit einer Fernsteuerung ausstattet. In beiden Fällen benötigt man Software die Entscheidungen trifft. Vielmehr sind es unterschiedliche Umsetzungen. Ich wage mal die These, dass die Variante mit der analogen Fernsteuerung die bessere ist. Man kann damit die Micromouse sehr viel kompakter bauen und hat gleichzeitig auf der Workstation mehr Möglichkeiten große Programme zu fahren. Machen wir uns nichts vor, will man heutzutage in Python programmieren und will man mit Simulationen arbeiten benötigt man viel Rechenleistung. Und nach wie vor sind Desktop PC die einzigen Geräte die das leisten. Desktop PCs sind wie der Name schon sagt, schwere Geräte die viel Strom verbrauchen. Man muss sich also entscheiden.

Eine Fernsteuerung ist aus zwei Gründen interessant. Erstens, kann man sie analog realisieren, also ohne CPU und zweitens kann man sie über einen human-operator steuern. Fragen wir dochmal die Händler was sie als Micromouse verkaufen. Einmal gibt es klassische Robotik-Händler welche darunter einen Roboter verstehen. Also eine Platine mit ATMega32 Microcontroller was 100 US$ kostet, mit dem entsprechenden Programmiergerät kommt man auf 120 US$. Es gibt jedoch auch Remote controlled Micromäuse, das sind keine Roboter und werden für 15 US$ das Stück verkauft. Diese werden mit einer Fernbedienung gesteuert. Sie werden meist als Spielzeug für Katzen verkauft, wo der Tierbesitzer damit sein Haustier bespasst. Das interessante ist, dass letztere bei den offiziellen Micromouse Wettbewerben nicht verwendet werden. Es gibt dort keine Scherzbolde die mit so einem Teil ins Labyrinth vorrücken, sondern 100% aller Teams verwenden Microcontroller-Basierende Roboter.

Lediglich bei Robocup gibt es eine eigene Liga wo passive Fußballroboter am Start sind. Die werden über externe Kameras getrackt und enthalten keine eigene Logik.

7.3 Remote Controlled Micromouse

Heute durchgeführte Roboterwettbewerbe sind noch viel zu stark auf Elektronik fixiert. Üblicherweise geht es bei Micromouse und anderen Challanges darum, 16bit Microcontroller zu programmieren und damit Aufgaben zu lösen. Die Idee lautet, dass diese Fähigkeiten wichtig seien und man so etwas über Technik lernt. Es gibt jedoch eine Alternative dazu. Und zwar eine die preiswerter ist. Sie besteht darin, dass man die rund 150 US$ teuren Microcontroller-Roboter durch Remotecontrolled Micromäuse ersetzt. Diese gibt es für 15 US$ als Elektrospielzeug in der Zoohandlung und haben sogar ein aufgenähtes Fell. Normalerweise werden Remote controled Micromäuse von Katzenbesitzern gekauft, sie enthalten intern gar keinen Microcontroller sondern werden analog gesteuert. Im Grunde muss man an die Fernbedienung nur einen Arduino anschließen und schon kann man das Spielzeug mittels Computer steuern. Der Vorteil ist, dass man einerseits riesige Programme ausführen kann, gleichzeitig aber die Elektronik minimalistisch ist.

Das Problem mit heute durchgeführten Micromouse Wettbewerben ist, dass man sich da schnell verzettelt. Um die üblichen Microcontroller-gesteuerten Systeme online zu bekommen muss man sich auskennen mit Hardware-Baastelei, Assemblerprogrammierung, C-Programmierung und Kleinstbetriebssystemen wie sie auf Microcontrollern verwendet werden. Ohne dieses Wissen wird man scheitern. Nur, das Wissen ist überflüssig, man braucht es nur, wenn ein echter Microcontroller verwendet wird. Es erhöht die Komplexität unnötig. Zeit die dafür aufgwendet wird, fehlt bei der eigentlichen Programmierung. Besser ist es, auf Microcontroller zu verzichten und sich stattdessen mit Softwareentwicklung zu beschäftigen. Die so erstellten Micromäuse sind sehr viel effiizienter.

7.4 Anziehungskraft von Forth

Warum auch heute noch Forth bei vielen beliebt ist hat mit der vermeintlichen Einfachheit von Forth zu tun. Angesichts von immer kompler werdenen CPUs mit Milliarden von Transistoren, und angesichts von undurchschaubaren GUI Oberflächen wünschen sich viele ein einfaches minimalistisches System zurück was ihnen einen direkten Zugang zur Machine ermöglicht. Viele sehen in Forth ein solches System, was nicht nur Betriebssysteme sondern auch Registermaschinen überflüssig macht. Die Einfachheit von Forth wird an zwei Parametern gemessen, einmal dass eine Forth Programmierumgebung inkl. Betriebssystem nicht mehr als 5 kb benötigt und das eine Forth CPU nicht mehr als 1000 Transistoren benötigt.

Aber sind solche Systeme wirklich minimalistisch? Bekanntlich kann man in einem Hobbyprojekt auch eine minmale CPU zusammenbauen aus wenigen Transistoren die mit einer Assemblerartigen Programmiersprache angesprochen wird. Aber minimalistisch ist so eine Architektur ganz sicher nicht. Ganz im Gegenteil versucht man solche Computer zu verkaufen will sie niemand haben. Sie sind zu teuer. Der Käufer müsste sich nicht nur mit der Verdrahtung der CPU auseinandersetzen sondern er müsste auch Assemblersprache lernen. Viel einfacher ist es für ihn, wenn er sich einfach einen Standard-PC kauft dort WIndows draufspielt und fertig. Das Problem besteht darin wie man EInfachheit misst. Allein der Codeumfang und die benötigte Zahl an Transistoren ist kein guter Maßstab, sondern man muss die Projektkomplexität anhand von ökonomischer Faktoren bestimmen. Damit ist gemeint, dass eine CPU mit 10 Milliarden Transistoren einfach er ist als eine mit 1000 Transistoren.

Schauen wir uns ein Beispiel aus der Softwareentwicklung an. Einige trauern den frühen Zeiten von Linux hinterher wo der komplette Kernel auf einer 3,5 Zoll Diskette Platz fand. Wollte man versuchen mit diesem Kernel ein heutiges System zu booten wird man sehen, dass es technisch gehen mag, aber dass es sehr teuer wird. Weil man sehr viel lesen muss, Erweiterungen programmieren und selbst dann wird die Leistung niedriger sein, als wenn man einen aktuellen Kernel bemüht. Rein nominell sind heutige Linux Kernel sehr viel umfangreicher als ihre Pendants in den 1994'er, trotzdem wurde dadurch effektiv Komplexität reduziert.

Die Zukunft liegt darin, dass nominell die Systeme an Umfang zulegen, aber qualitttiv umso überschaubarer werden.

7.5 Neoludditen träumen von Forth

Die technische Weiterentwicklung aufzuhalten ist simpel: und zwar muss man einfach in die Vergangenheit blicken und jene Dinge hochhalten, die damals relevant waren. Wenn man Computer aus den 1980'er Jahren verwendet die möglichst wenig Hauptspeicher und noch gar kein Betriebssystem hatten ist das der beste Schutz gegen Künstliche Intelligenz. Es gibt auf Youtube viele Bastelprojekte wo Leute die früher auf C-64 Treffen und Forth-User-Treffen aktiv waren jetzt damit anfangen ihr Wissen in die Gegenwart zu übertra-

gen. Das bedeutet konkret, dass alles abgelehnt wird was modern und zukunftsorientiert ist und stattdessen man sich aus eigenen Chips und selbst erstellter Software einen Roboter baut. Also schön mit Microcontroller, selbst programmierten Betriebssystem und am besten noch selbst-geätzten Leiterbahnen. Damit wird ein ganz besonderer Retro-Style erzeugt wo man im Jahr 2017 jene Technologie kopiert die in den 1980'er erfunden wurde und das mit heutigem Wissen aufwertet.

Dieser Retrostyle hat mehrere Gründe. Einmal versuchen die Bastler darüber Komplexität zu senken. So nach dem Motto: wenn der RAM des PC nur 30 kb groß ist, ist es leichter den Computer zu verstehen. Zweitens wird das Fachwissen aus der Vergangenheit als wertvoll erachtet und zu guter letzt ist es ausgeschlossen, dass einer der so programmierten Roboter echte Intelligenz zeigt. Die wichtigste Eigenschaft von Retro im allgemeinen und Forth im Besonderen ist, dass die Projektkosten damit in die Höhe gehen. Wer nochmal sein eigenes Betreibssytem und seinen eigenen Compiler from scratch entwickelt der kann sich jahrelang damit beschäftigen ohne jemals etwas konkretes zu realisieren. Man bastelt so vor sich hin und schafft sich Räume die man leicht überblicken kann. Das hier ist der Befehl um etwas auf den Stack zu pushen, dass hier ist der 9v Servomotor und das da, ist mein selbsterstellter Editor in 50 Lines of Code. Alles Klar, die Welt ist unter Kontrolle.

Dazu gibt es natürlich eine Gegenbewegung. Diese ist nicht auf konkrete Computer oder konkrete Programmiersprachen fixiert sondern versucht die Projektkosten zu senken. im Zentrum steht der Wunsch einen Roboter zu entwickeln der möglichst in einem Wettbewerb gewinnt und zwar mit möglichst wenig Aufwand. Die konkrete Technologie ergibt sich aus dieser Prämisse. Sie lautet, dass man vorhandene COTS Systeme nutzt, also Standard-Notebook, Standard-Linux und das man den Hardwareteil des Roboters möglichst reduziert. Das heißt, man verwendet ein RC Car, modifiziert dort den RC Transmitter um ihn via Arduino anzusteuern und installiert im Raum mehrere Kameras um eine Virtual Realiy Umgebung herzustellen. Dann wird Unity3D genutzt um darin Scripte zu starten die den Roboter ins Ziel bringen. Das wäre die Lowcost Version zur Robotersteuerung. Mit Forth oder Hardwarebastelei hat es nichts mehr zu tun, sondern hier geht es um richtige Robotik.

8 Challanges

8.1 Der Micromouse Wettbewerb

Nach meiner Recherche ist der Micromouse Wettbewerb genau ein Mittelding zwischen Mainstream Informatik und Forth Community. Er ist deshalb Mainstream weil dort C-Programmierer aktiv sind, aber aus Sicht von Forth ist es interessant weil einige der Microcontroller so kompakt sind, dass nur Forth darauf läuft und sonst gar nichts. Die wichtigste Frage ist natüriich wie man bei einem solchen Wettbewerb gewinnt, und zwar stellen sich diese Frage sowohl C-Programmierer als auch Forth Leute gleichermaßen. Üblicherweise lautet die Antwort darauf, dass es auf den richtigen Algorithmus ankäme, also ein Floodfill-Algorithmus, ein Mapping Algoirthmus oder ein Verfahren um die Sensoren auszulesen. Meine Beobachtung ist hingegen, dass die Algorithmen fast egal sind, sie sind austauschbar und es ist schwer den besten zu ermitteln. Worum es eher eher geht ist Software-Engineering, also wie man mit begrenzter Manpower die Anforderungen des Wettbewerbes erfüllt und wie man vorgeht wenn die Software einen Bug enthält.

Mit diesem Mindset erkennt man, dass Micromouse eben kein C-Programmierwettbewerb und erst recht kein Forth Contest ist. Der Grund warum sich einige dennoch darauf fokussieren ist weil sie es nicht anders gelernt haben. Sie glauben, dass man sich intensiver mit der Hardware beschäftigen müsste und dass man stundenlang mit dem Lötkolben an der Jtag Schnittstelle herumdoktern müsste. Andere glaube, es ginge darum den Compiler optimal auszunutzen und jedes Byte zu verwenden was der RAM hergibt. Dabei ist die eigentliche Stelle worauf man seine Energie konzentrieren sollte eher ungewöhnlich für diesen Wettbewerb: es ist natürlich die Software.

Will man Micromouse richtig spielen, so wird man dazu objektorientierte Programmiersprachen, Issue-Tracker und UML Diagramme benötigen. Gerade weil es sich um eine Embedded Competition handelt muss man da so vorgehen als würde man für ein Unternehmen ein Java-Framework entwickeln. Die benötigten Fähigkeiten um auf Platz 1 zu kommen sind angesiedelt zwischen Projektmanagement, Wissenschaftlicher Quellenrecherche und Java Programmierung. Mit diesem Mindset kann man dann ganz am Ende und als nebensächlichste Aufgabe von allem den fertigen Assembler Code auf den Microcontroller draufspielen oder alternativ den Forth code programmieren. Ich würde mal schätzen, dass der reine Code um den Microcontroller zu steuern in weniger als 50 kb hineingeht. Entweder hocheffizienter Forth-Code oder per C-Compiler assemblierter Hochsprachencode. Um jedoch diese 50 kb zu schreiben muss man vorher dutzende Prototypen im Simulator getestet haben, man muss Literatur ausgewertet haben und man muss mit Java UML Diagramme gezeichnet haben. Anders geht es nicht bzw. artet aus in Rumprobieren.

Schaut man sich einmal erfolgreiche Teilnehmer der Challange an, so gibt es da nicht den einen Algorithmus der auf magische Weise den Roboter steuert, sondern üblicherweise wurden sehr viele Detailfragen in Software geklärt. Angefangen von der Einschaltprozedur, das heißt das nach dem Druck auf den Button das Teil wirklich losfährt, über die Wegeplanung, das Navigieren in Kurven, spezielle Heurstiken für Highspeed Fahren bis hin zur Objekterkennung. Am ehsten kann man diese Verfahren noch unter dem Begriff des Scripting AI zusammenfassen, bzw. subsumption Architektur. Etwas genauer gesagt geht es darum, die Domäne Micromouse in Codezeilen zu überführen. Das ist ungefähr das, was die Teams dort machen.

8.2 Projektdenken anstatt Forth Programmierung

Welche Kernkompetenzen sind erforderlich um bei der Micromouse Challange erfolgreich zu sein? Auf den ersten Blick sieht es danach aus, als ob es um Programmieren eines Microcontrollers ginge. Folglich wäre Forth die beste Methode. Aber ist Micromouse wirklich ein Programmierwettbewerb? Anfangs war er das vielleicht mal, aber schaut man sich einmal an, welche Teams heute daran teilnehmen und welche best-practice Methoden eingesetzt werden so haben sich die Anforderungen verändert.

Nicht etwa dass sie komplexer geworden sind, ganz im Gegenteil. Im Laufe der Jahre fand beim Micromouse Wettbewerb eine Suche nach einem Minimalismus statt. Die Frage ist, was der geringstmögliche Aufwand ist um den Roboter gerade so ins Ziel zu bringen. Anders gesagt geht es darum, dass die Teilnehmer ihre Kosten senken, also der Aufwand den sie treiben müssen. Man kann sagen, dass es bei Micromouse darum geht den billigsten Roboter zu bauen. Billig muss nicht unbedingt bedeuten, dass sein RAM nur maximal 10 kb groß sein darf, Billig kann bedeuten, dass man einen ausgewachsenen Laptop mitsamt Onboard Linux und Wifi an den Start bringt. Sondern es geht um die Gesamtkosten die das Projekt erzeugt, also wieviel Stunden man braucht um ein gewünschtes Ergebnis zu erzielen.

Die Programmiersprache Forth ist besonders stark auf die Programmierung als solche fokussiert. Also darauf, wie man bare-metal die Probleme löst, und vor allem auf Meta-Programmierung. Also wie man Bootstrapping Verfahren durchführt. Das heißt, man schreibt sich zunächst einmal einen Interpreter für eine Domänenspezifische

Sprache und entwickelt das zu einem Robot-control-System weiter. Dabei fängt man ganz unten an bei der Stackmanipulation und arbeitet sich dann langsam hoch. Aus Sicht der Informatik ein guter Ansatz, nur leider wird man damit bei Micromouse scheitern. Es sind nicht jene Fähigkeiten auf die es ankommt. Sondern wie man den Wettbewerb eher gewinnt ist wenn man einen Bugrracker aufsetzt, UML Diagramme zeichnet, Literatur recherchiert und Leute ins Team holt die sich mit Marketing auskennen. Anders formuliert, Micromouse hat mehr Ähnlichkeit mit einem Business-Projekt was in einem Unternehmen ausgeführt wird und wo es Management-Entscheidungen als mit einem lowlevel Hacking Projekt.

Ich glaube, dass Forth und vielleicht sogar C beim Micromouse Wettbewerb nichts verloren haben. Es sind Sprachen die veraltet sind. Sie sind zwar nominell klein und schlank aber nicht im Sinne eines schlanken Micromouse Projektes. Das heißt, wenn man eine Programmier-orientierte Herangehensweise wählt ist die Gefahr groß dass man beim Wettbewerb scheitert. Besser ist es, in größeren Dimensionen zu denken und mit einer UML zentrierten / Projektorganisation an das Thema heranzugehen und sich nach dieser Zielsetzung für konkrete Technologien entscheiden. Wenn es Microcontroller mit vorinstalliertem Linux gibt, dann sollte man sie einsetzen. Sie erleichtern die Dinge ungemeint. Wenn es Librarys bei github gibt, nur her damit. Wenn man irgendwo einen Robotersimulator in 2D Auftreiben kann umso besser. Man sollte Micromouse eher aufziehen wie ein Großprojekt, also wo man erstmal Spitzenprogrammierer aus den USA einfliegen lässt, die vor dezidierte Workstations setzt und wenn es dann wissenschaftliche Probleme gibt werden die mit Mathematikern, Soziologen und Künstlern ausdiskutiert.

Rein formal ist die Software um einen Roboter durchs Labyrinth zu schicken erstaunlich kompakt. Wenn man sich Mühe gibt kann man den Assemblercode in weniger als 50 kb unterbringen. Um jedoch diesen Code zu schreiben reicht es nicht aus, sich in ein Forth System einzuloggen und auf der Textkonsole einige Words einzugeben. So bekommt man den Code nicht erstellt. Das darf eher als Antipattern bezeichnet werden. Sondern Sourcecode wird in dezidizierten Programmierprojekten erstellt, wo man also eine lernende Organisation aufbaut und Strukturen schafft damit diese nicht nur den Code schreibt sondern sich auch das nötige theoretische Wissen aneignet.

Leider sind klassische Lehrangebote an den Universitäten noch auf eine reine Programmierausbildung fokusiert. Wo also anhand von konkreten Programmiersprache ein Maze-Solving Algorithmus vorgestellt wird und der dann zur Übung auf einem Roboter implementiert wird. Wer so an die Sache herangeht hat bereits verloren. Das Problem lautet, dass ein 100 Zeilen Programm nicht ausreicht für einen Roboter. Es geht nicht um das Programmieren als solches sondern um das Erfinden von Algorithmen, das beseitigen von Fehlern und das Dokumentieren der Software. Anders formuliert, wenn man Micromouse richtig angeht, ähnelt es eher dem Schreiben einer Diplomarbeit, wo man also das Thema untersucht und Programmieren darin nur eine Nebenbedeutung besitzt.

Der Witz ist, dass die Standard-Algorithmen und die Standardprogrammiersprachen alle verfügbar sind. Wer heute from Scratch einen Micromouse Roboter bauen will, hat eigentlich alles was er benötigt: Compiler, Interpreter, Programmiersprachen, Assembler, Pathplanner, Vision Systeme, High-Level-Planner, PDDL, 16 bit Microcontroller usw. Die Kunst besteht darin, dass alles zu komibinieren zu einem lauffähigen ganzen und außen herum auch noch das Team bei Laune zu halten. Das ist die eigentliche Kernkompetenz auf die es ankommt.

Lines of Code Warum sich nicht die Sprache Forth sondern die viel komplexere Kombintation aus C plus Unix hat durchsetzenk können hat einen simplen Namen: lines of Code. Damit ist gemeint, dass die Anforderungen an Software gewachsen ist und die einzige Möglichkeit mit den Bedürfnissen der User mitzuhalten besteht darin, wenn man die Codezeilen erhöht. So besteht der Linux Kernel heute bereits aus 20 Mio Codezeilen, Tendenz steigend. Für Computerspiele und Internetbrowser gilt das selbe. Das mag jetzt etwas überraschen weil nachdem man Linux startet eigentlich nicht viel passiert, es laufen einige Statusmeldungen über den Bildschirm und das wars auch schon. Man könnte diese Meldungen gut in einer 2 kb große Zip Datei unterbringen und den Sourcecode von Linux beträchtlich einstampfen. Die Schwierigkiet besteht darin, dass kleine schlanke Programme nicht wirklich klein und schlank sind. Wenn man einen Forth Interpreter in 2 kb vorliegen hat ist das zwar formal gesehen ein Betriebssystem inkl Programmiersprache defakto ist es für den user damit nicht möglich seine Ziele zu erreichen. Es ist nicht möglich, durch kleinere Modifikationen des Forth Interpreters ihn dazu zu bringen, Webseiten anzuzeigen oder Spiele zu spielen. Es hat seinen Grund warum reale Projekte aus vielen Codezeilen bestehen. Schon kleinere Computerspiele kommen auf 1 Mio Lines of Code. Und es gibt bis heute keine Methode wie man das ändert. Ganz im Gegenteil, als Anfang der 1990'er die objektorientierte Programmierung erfunden dann nur zu dem Zweck die Anzahl der Codezeilen weiter zu erhöhen. Es reichte nicht aus, einfach nur in MS-DOS irgendwelche Spiele zu haben, sondern die Anwender wollten ab den 1990'er Mehrfenster-Betriebsssysteme mit komfortablen Buttons und verschachtelten Pulldown Menüs.

Leider wurde Forth nicht dafür geschaffen große Codemengen einzugeben oder zu verwalten. Das beginnt mit der reverse polish Notation und zieht sich über fehlende Array Funktionen weiter. Durch die Abwesenheit von Standards fehlte es auch an Bemühungen die ein Betriebssystem zum Ziel hatten, wodurch es unter Forth keine leistungsfähigen IDEs wie den Emacs gibt. Kurz gesagt, mag in den 1970'er Jahren noch weitestgehend Gleichstand zwischen Forth und C gegeben haben (beide Systeme waren extrem kompakt und es fehlte an Bibliotheken) hat sich die Lage zuungunsten von Forth entwickelt. Forth wird diesen Rückstand niemals aufholen können. Die Menge an Code die für C/Java/C++ usw. produziert wird ist es um einiges größer was an Forth Code neu geschrieben wird.

Forth ist keineswegs uninteressant als Sprache. Unter den Toylanguages gehört sie zur Nr. 1 und es ist lehrreich sich damit zu beschäftigen. Nur, in erster Linie erfüllen Computersprachen einen Zweck. Die Entwicklung der Software verläuft Usergetrieben und hier kann Forth nicht mithalten. Es gibt schlichtweg keine User die an Forth Programmierer herantreten und Änderungen haben wollen. Sondern die User wollen professionelle Software auf ihrer Hardware haben.

Die Schwierigkeit mit Forth Sourcecode ist, dass er im Zweifel keineswegs kleiner ist, als vergleichbarer Code in C. Der selbe Algorithmus einmal in C und einmal in Forth implementiert nimmt ungefähr den selben Platz ein. Schon die Sprache Python ist mit dem Anspruch gescheitert, die Anzahl der Codegröße zu reduzieren. In der Praxis kann man zwar mit Python rund 10% weniger Zeilen nutzen und trotzdem das gleiche tun, aber ein Projekt mit 1 Mio Lines of Code reduziert sich davon nur minimal. Bei Forth gilt gleiches. Wenn man sich bemüht kann man mit etwas Übung die Words nebeneinander und nicht untereinander schreiben, aber den Codeumfang wird man dadurch nicht reduzieren. Auch Forth ist eine prozedurale Programmiersprache, was bedeutet, dass man Unterprogramme schreibt und nach dem Teile und Hersche Paradigma die angeht.

8.3 Es gibt zuwenig Challanges

Die akademische Informatik allgemein aber auch Forth im Besonderen haben ein Problem: es wird nicht unterschieden zwischen Sinn und Unsinn. Genauer gesagt gibt es jede Menge Theorie darüber wie Computer funktionieren und wie man sie programmiert aber

nur wenig belastbare Fakten. In dieser Atomosphäre des anything goes blühen natürlich auch Pseudowissenschaftliche Theorien die wissenschaftlich auftreten aber keinen Wert besitzen. Um diese zu identifzieren sind Coding-Challanges die beste Methode. Interessanterweise gibt es gerade zu den Kernthemen keine solche Challanges. Ich habe mal speziell im Bereich Forth nach einer Coding-Challanges gesucht: Fehlanzeige. Und Forth wird gerade im Bereich der akademischen Informatik als letztes Bollwerk verstanden hinter dem man sich verschanzt und mit dessen Hilfe man abgesicherte System entwerfen will und die Computer von morgen. Eigentlich wäre eine Forth Coding Challange sehr simpel: es gibt eine Aufgabe, wie schreibe einen Primzahlgenerator in einer 1 Stunde und dann schaut man wer der beste Programmierer ist und wie die Lösung aussieht. Solche Challanges werden nicht veranstaltet. Auch bei Micromouse wo man Forth gut einsetzen könnte, fehlt es ausgerechnet an Forth Projekten. Nicht viel anders sieht es bei einem anderen Thema der Informatik aus, wo sich viele Leute an den Hochschulen mit beschäftigen: Neuronale Netze. Auch dort wird viel darüber geschrieben auch sehr neue und avant-Garde Forschung. Komischerweise fehlt es ähnlich wie bei Forth jedoch an Programmierwettbewerben. Wo man also ein neuronales Netz aufsetzt und das dann konkrete Dinge tun soll. Ein guter Anfang besteht darin, eines zu programmieren was Primzahlen lösen kann ...

Das Fehler solcher dezidierten Wettbewerbe lässt sich nichts ahnen. Vermutlich werden sie deshalb nicht durchgeführt weil man insgeheim Angst hat vor den Resultaten. Weil plötzlich das Buchwissen was die Akademiker seit 30 Jahren angesammelt haben auf dem Prüfstand steht und wenn ein Student die Aufgabe nicht lösen kann, ist womöglich nicht der Student daran schuld, sondern Neuronale Netze sind doch keine guten Mustererkenner wie es immer heißt. Ich habe nichts gegen Forth oder neuronale Netze, sondern ich möchte das wissenschaftliche Umfeld kritisieren indem sie unterrichtet werden. Es macht keinen Sinn nur theoretisch zu Themen der Informatik zu forschen und gepeerreviewte Paper bei Elsevier zu veröffentlichen wenn es keine Möglichkeit gibt, Theorien zu wiederlegen. Das interessante bei Programmieraufgaben ist, dass es dort eben nicht darum geht etwas zu verstehen und sich mit jemand anderem der es ebenfalls verstanden zu unterhalten, sondern bei Programmierwettbewerben geht es um eine soziale Konkurrenzsituation bei dem Verlierer produziert werden. Also Teilnehmer, die es nicht geschafft haben und die versuchen dafür eine Ausrede zu finden. Dieser Reflektionsprozess ist wichtig, nicht so sehr für den einzelnen, sondern für das Fach insgesamt. Darüber wird es möglich schneller zu erkennen, welche der geschätzt 100000 Paper die über neuronale Netze geschrieben wurden, Unsinn enthalten und welche nicht. Aktuell sind alle gleich gerankt und der einzige Bias bei Google Scholar ist die Zitierhäufigkeit. Wenn aber alle den Kaiser wegen seiner Kleidung loben wird niemand es wagen zu widersprechen. Die Informatik ist auf Mittel der Selbstreflektion angewiesen, nur so kann man steuern wohin die Reise gehen soll

Schauen wir mal an, wie derzeit über das Thema Neuronale Netze diskutiert wird. Üblicherweise lädt sich jemand einen Neuronalen Netz SImulator aus dem Netz herunter, stellt dort ein er möchte 3 Hiddenlayer und 10 Eingangsneuronanen haben, trainiert ein wenig mit Beispieldaten und stellt fest, dass der Fehlerwert sich nicht weiter senken lässt sondern einfriert. Wenn er jetzt bei Stackoverflow nachfragt warum sind die Antworten entlarvend. Der erste meint, dass der Fragesteller das Thema nicht richtig verstanden hätte, der nächste meint die Frage sei zu allgemein gehalten und er bräuchte weitere Informationen über die Software, der nächste glaubt zu wissen, dass man die Lernrate irgendwo einstellen kann und der letzte hat überhaupt keinen Plan kennt aber ein gutes Buch was er empfehlen kann. Nur, wirklich vorangekommen ist dabei niemand. Im Grunde war das ganze ein unverbindliches Gespräch was aufgezeigt hat, dass es offenbar niemand gibt der sich näher damit beschäftigt hat. Würde man hingegen einen Wettbewerb durchführen wo alle 10 Teams an der konstanten Fehlerrate scheitern, die sich nicht weiter verbessert lässt hätte die Informatik ein echtes Problem. Jetzt könnte man nähmlich mit der Ursachenforschung beginnen und die Theorie der neuronalen Netze an sich hinterfragen. Genau dieser schonungslose Umgang mit Realität ist derzeit eher die Ausnahme.

9 Automaten

9.1 Registermaschinen aka Random Access Machinen

Ein Forth Chip basiert üblicherweise darauf, dass es einen Instruction Pointer gibt, welcher den nächsten Befehl spezifziert und einen Stack indem sich Daten befinden die verarbeitet werden. Häufig wird gesagt, das müsse so sein, und es sich um das kleinst-mögliche Berechnungsmodell handeln würde. Also das Standard-Verfahren um eine CPU zu beschreiben. Das ist so nicht korrekt. Eine Weiterentwickelte Form der Stackmachine ist eine Registermaschine, dort gibt es neben dem Befehlscounter auch einen Speichercounter. Genauer gesagt kann die CPU sowohl auf Befehle als auch auf Daten wahlfrei zugreifen. Mit wahlfrei ist gemeint, nicht nur das oberste Element eines Speichers ansprechen sondern beliebige Elemente. Und hier nähern wir uns den heute verbreiten Mikroprozessoren welche ganz überwiegend als Registermaschine ausgelegt sind.

Um sich den Unterschied vor Augen zu führen kann man sich ein Python Array vorstellen. Einmal die Variante, bei dem man ganz normal darauf zugreift also “print temp]2]” und dann eine eingeschränkte Version bei dem man nur jene Elemente lesen / schreiben kann die sich am Ende befinden “print temp[-1]”. Schaut man sich reale Python Programme an, so wird man sehen, dass die Variante mit dem wahlfreien Zugriff beliebter ist, sie lässt sich komfortabler benutzen. Aber, grundsätzlich kann man auch nur mit einem Stack arbeiten. Das ist jedoch anspruchsvoller zu programmieren.

Stack vs Register In einem Punkt sind sich Stackbased Machines und Registermachinen ähnlich. Sie haben beide einen Programmcounter. Damit ist eine Speicheradresse gemeint, an der der nächste Befehl wartet. Auf welche Daten ein Opcode dann zugreift, ist unterschiedlich. Bei einer Stackbased-CPU kann der Befehl nur auf den Stack zugreifen und dort nur auf das oberste Element. Bei einer Registermaschine hingegen auf praktisch jede Stelle im Speicher. Der Speicher wird jeweils in die Register eingeblendet und wenn das nicht funktioniert gecached.

Aus Sicht der theoretischen Informatik ist die egal ob man eine Stackmachine oder eine Registermachine verwendet. In beiden Fällen lässt sich ein Programm abarbeiten. Der Unterschied besteht jedoch aus Sicht des Engineering, also wenn man einen möglichst effizienter Computer bauen möchte. Hier ist die Stackmachine im Vorteil. Eine Registermaschine ist nur die zweitbeste Möglichkeit eine Turing-Maschine in Hardware zu realisieren. Es ist prinzipiell möglich aber die Ausführungszeiten sind erhöht.

Der Grund warum sich im Mainstream die Registermaschine durchgesetzt hat, liegt darin begründet, dass man dort leichter C-Compiler für schreiben kann. Es gab zwar Versuche auch für Stackmachinen C-Compiler zu entwickeln doch ohne Erfolg. Am effizientesten lassen sich diese Maschinen in Forth programmieren und üblicherweise wird der Code manuell programmiert.

Üblicherweise grenzt sich die Forth-Community von der übrigen Informatik scharf ab. Doch so groß sind die Unterschiede gar ncht.

Sowohl die Java Virtual Maschine als auch die Python Virtual Maschine sind als Stackmachine implementiert. Um sie auf einer Registermaschine auszuführen benötigt man einen Interpreter. Es ist also keineswegs unmöglich zwischen Stack- und Registermachine hin- und herzuübersetzen. Wenn man Python Sourcecode in Python Bytecode übersetzt, wird eine objektorientierte Hochsprache in eine Lowlevel Stackmachine konvertiert, und wenn man Python Bytecode auf einer x86 CPU ausführt, wird eine abstrakte Stackmachine gegen eine x86 Registermachine gemappt.

Der Grund, warum Java und Python sehr junge Plattformen sind hat exakt mit dieser Übersetzung zu tun. Üblicherweise wurde bevor es Java gab, so gearbetiet: der Programmierer hat seinen Code in C/C++ geschrieben, dieser wurde von einem Compiler in Maschinencode übersetzt und zwar für eine Registermaschine. Bis heute kann gcc beispielsweise nicht für Stackmachinen Code erzeugen. Die Aufgabe gilt als zu anspruchsvoll und auch C to Forth Compiler gibt es nicht.

Dann kam Java und hat alles verändert. In dem Moment wo man sich für eine virtuelle Maschine entscheidet erhält man 2 Layer: einmal die Programmiersprache Java als solche, also schön objektorientiert mit mächtigen Bibliotheken und darunter dann die Laufzeitumgebung die sehr systemnah programmiert wird. Und die Java-Runtime-Engine war dann auch der Ort wo an erstmalig leistungsfähige Stackmachine to Registermachine Compiler entwickelt wurden. Als Input wurde anders als bei früheren Compilern nicht etwa C oder C++ verwendet, sondern der Bytecode der schon da war. Und der wurde dann in einem zweiten Schritt auf x86 Registermaschinen gemappt.

Es wäre durchaus möglich, eine JVM für einen Stackprozessor zu entwickeln. Das gleiche gilt für Python und für C#. Das heißt, man würde den Bytecode aus einer virtuellen Stackmachine auf eine physische Stackmachine mappen.

Ich glaube, auch in Zukunft wird es keine C to Forth Compiler geben, die Aufgabe ist zu anspruchsvoll. Aber was durchaus denkbar wäre, das ist eine JVM für den GA144 Arraycomputer. Also eine Engine welche Java Code auf einem Forth prozessor ausführt.

Hier kommen wir zu der grundsätzlichen Motivation warum Java erfunden wurde. Java ist nicht einfach nur eine Programmiersprache, sondern Java zerteilt einen klassischen Compiler in zwei Ebenen. Java->Bytecode und Bytecode->Maschinecode. Dadurch wird die Komplexität drastisch gesenkt und man kann jeden Layer einzeln optimieren. Aktuell ist es so, dass alle Virtuellen Maschinen (Java, Python und C#) aussschließlich für Registermaschinen x86 Code erzeugen. Vorstellbar wäre es aber, wenn man die Virtual Maschine aufteilt. Ein Backend für x86 Maschineinstructions und eines für Forth Chips. Damit hätte man dann eine extrem leistungsfähige Pipeline wo die Softwareentwickler in Java programmieren können und der Code dann auf Forth CPUs ausgeführt wird.

Mit der traditionellen Compilertechnologie a la C++ ist sowas nicht möglich. Ein solches Projekt wäre gigantisch, es ist nicht handelbar. Wenn man es jedoch mit Java im Hinterkopf betreibt könnte es klappen. Die Anforderungen ein JVM für einen Forth Backend zu schreiben ist überschaubar.

Das läuft darauf hinaus, eine Programmiersprache zu entwickeln die schneller abgearbetiet wird als C. Technisch ist das machbar mit der oben beschriebenen Methode. Aber, auch heute schon kann man in speziell entwickelten FPGA Schaltungen etwas ähnliches machen. Wo man also auf eine Programmiersprache verzichtet und den Algorithmus gleich als IP Core formuliert. Wenn man den ausführt kann man jedes C-Programm übertrumpfen. Das ganze ist also weniger ein konkretes Problem was sich stellt sondern gibt eher die Richtung vor wohin sich die Computertechnik entwickelt. Das heißt, es ist vorstellbar, dass JVM Plattformen weiter verbessert werden und dass die CPUs ebenfalls sich mehr auf diesen Trend werden einlassen. In der letzten ARM Generation gibt es beriets ein Bit was man aktivieren kann um Java ByteCode nativ auszuführen. Laut einer Messung war der Effekt gering (10% schnellere Codeausführung). Wenn man das ganze weiterentwickelt, dürfte sich da noch mehr Performance rausholen lassen.

Das Problem ist ein anderes: letztlich sind diese Steigerungen nichts neues. Sondern das Moorsche Gesetz sieht ja gerade vor, dass sich jedes Jahr die Leistung verbessert. Und der Wechsel von Register auf Stackmachinen plus die dazugehörigen Compiler sind nur ein Spielfeld unter vielen.

Reizt man das alles aus, bleibt alles beim alten. Das heißt, vielleicht hat man irgendwann eine JVM die auf einem GA144 Forthprozessor mit 1 Watt in hoher Geschwindigkeit läuft und man kann trotzdem in Java darauf programmieren. Aber wen interessiert das? Wenn man keinen Java Code hat, nützt die verbesserte Hardware gar nichts.

Letztlich ist also Java und Forth, Stackmachinen und Registermaschinen alle ein und dasselbe. Nicht auf der technischen Ebene wie es im Detail funktioniert, sondern auf einer kulturellen Ebene. Es geht im Kern darum, Maschinen effizienter zu machen, CPUs zu entwickeln und Code auszuführen.

9.2 Stackmachine vs. Registermachine

Im Jahr 2016 ist ein Peper erschienen was virtuelle Stackmaschinen mit virtuellen Registermaschinen vergleicht [3] und zu dem Schluss kommt, dass Registermashinen schneller sind (um 30%). Speziell für diesen Vergleich wurden zwei Virtuelle Maschinen in Software programmiert, und diese auf realer Hardware ausgeführt. Es ging konkret um die Frage, ob die Java-Virtual Maschine (welche als Stackmaschine aufgebaut ist) nicht durch eine Registermaschine ersetetzt werden sollte.

Das interessante an dem Vergleich ist, dass er für sich betrachtet korrekt ist, aber mit sehr eingeschränkter Sichtweise das Thema bearbeitet. Ein fairer Vergleich sieht so aus, dass man anhand der Performance per Watt Benchmarks einen Vergleich anstellt. Denn, um eine Registermaschine in Hardware zu realisieren braucht man mehr Transistoren als wenn man eine Stackmachine in Hardware realisiert. Wenn man den Vergleich nach dieser Maßgabe ausführt ist das Ergebnis eindeutig zugunsten der Stackmaschinen. In mehreren Powerpoint Vorträgen zum GA144 Chip wurde die Performance per Watt Maßzahl verwendet.

Anders ausgedrückt, es ist technisch möglich eine Registermaschine zu bauen, die effizienter ist als eine Stackmaschine, sondern um sowas zu “beweisen” muss man schon ziemlich tief in die Trickkiste greifen was mit Wissenschaft jedoch nichts mehr zu tun hat, sondern eher mit Ideologie. Nein, das einzige Gegenargument was Stackmachinen spricht ist, dass es dafür keine Hochsprachencompiler gibt. Der GA144 Chip ist nutzlos, weil sich darauf kein Java Code ausführen lässt. Das ist ein faires Argument.

Die Nichtverfügbarkeit von Hochsprachencompilern ist jedoch keine technische Eigenschaft sondern hat etwas mit kulturellen Traditionen zu tun. Das man also irgendwie glaubt, es würde nicht gehen und es würde nichts bringen und deshalb in dieser Richtung nicht geforscht wird. Tatsächlich sind Stackcomputer die Zukunft, sie sind um einiges mächtiger. Insbesondere wenn man bedenkt, dass dort die nächsten Features wie Arrayprozessoren und clockless Ausführung schon heute als Prototypen realisiert wurden, während das bei Registermaschinen noch nicht geht und sehr viel komplizierter wäre.

Gehen wir etwas näher auf den GA144 Chip ein. Hardwaremäßig schlägt er Intel Prozessoren um Längen. Er verbraucht weniger Energie pro Operation, ist durch das Zusammenshalten von mehreren EInheiten nach oben skalisiert und kann deutlich preiswerter produziert werden. Im Grunde ist der GA144 Chip eine Art von Xeon

Phi, nur dass es nicht 3000 US$ kostet sondern nur 30 US$. Das sich der Chip nicht durchsetzt hat keine technischen Gründen sondern es sind Wettbewerbsrechtliche Fragen. Würde man sowas auf den Markt bringen, wäre Intel pleite.

Halt, nicht ganz so schnell. Einen Nachteil hat die GA144 CPU. Man muss sie zwingend in Forth programmieren. Das ist jene Programmiersprache die nachweislich (ich habe es selber versucht) sehr schwer zu erlernen ist. Forth ist eine Zumutung, insbedonere wenn man objektorientiertes Programmieren gewöhnt ist. Man kann in Forth durchaus Programme schreiben, aber das Coden, Testen und Debuggen geht um einiges langsamer. Ich würde mal schätzen, dass man für ein Forth Programm rund 100x länger benötigt als für das selbe Programm in Python bis man es programmiert hat. Und das ist in der Tat ein echter Nachteil.

Der Schlüssel um den Markterfolg von Forth Chips zu ermöglichen sind JVM Engines für Stackmaschinen. Also Systeme welche Java-Bytecode in Forth-Maschineninstructions übersetzen können. Soetwas zu programmieren ist anspruchsvoll aber möglich. Es ist wesentlich leichter als würde man einen Java-to-Forth Converter schreiben. Weil, der JVM Bytecode ist bereits für eine Stackmaschine optimiert, man kann ihn gut auf eine physische Stackmaschine mappen. Damit wird es möglich, die Performance Vorteile eines GA144 Chips an die Java-Programmier-Community hochzureichen. .Das man also in seiner Lieblingssprache ganz normal Klassen anlegt und Java-Libraryies nutzt, das dann aber auf einem Forth Chip ultraschnell ausführt.

Diese Technologie dürfte die Hardware-Industrie vor die selben Schwierigkeiten stellen wie Linux es mit Microsoft gemacht hat. Es wird ein hoher Vorteil für die Kunden entstehen während die Hersteller klassischer Hardware vor Probleme gestellt werden.

9.3 Über Kellerautomaten und Mealy Machinen

Computer waren schon immer hierarchisch aufgebaut. In dem Sinne dass auf der untersten Ebene Nullen und Einsen existieren und erst viele Ebene darüber dann Betriebssysteme und Software ins Spiel kommt. Einmal lässt sich diese Hierarchie historisch erklären, in dem Sinne dass es in den 1950'er noch keine GUI Betriebssysteme gab, aber auch wenn heute from Scratch neue Computer baut wird diese Hierchie sichtbar. Besonders schön kann man das am Beispiel von Forth betrachten. Ein Forth System ist von der Hardware sowie der Software ein Lowlevel Computer. Also ein System wo noch sehr vieles nicht existiert.

Die einfachste Methode um Computer zu bauen ist, wenn man auf Software komplett verzichtet und stattdessen die Ablauflogik direkt in Hardware realisiert. Hierbei spricht man von einer State-Machinen. Diese lassen sich direkt aus Logik-Baulelementen erzeugen. Auch State-Machines sind turing-mächtig und zwar weil man extrem komplexe Verzweigungen aufbauen kann die exakt das tun was ein normaler Computer auch kann. Die Frage ist nur: wer programmiert solche State-Machines? Und genau hier liegt die Notwendigkeit nach möglichen Alternativen zu suchen. Genau genommen diente die Weiterentwicklung der Computer seit seinen Anfängen dazu, dass Programmieren zu erleichtern. Konkret bedeutet es, dass Hardwaremäßig mindestens seit den 1930'er klar war wie man leistungsstarke Computer baut. Bereits damals gab es Logikbausteine die schnell genug schalten konnten. Die Schwierigkeit bestand darin, dass es keine Programme für diese Computer gab.

Wenn man in Hardware realisierte State-Machine um weitere Elemente erweitert wie einen Stack erhält man einen Kellerautomaten. Erweitert man diese um Register erhält man die Harvard Architektur, erweitert man diese um Microcode entsteht ein CISC Prozessor, dieser wird dann um ein BIOS, C-Compiler und Betriebssystem erweitert und irgendwann erhält man heute übliche PCs, die also nach dem Einschalten einen Webbrowser plus Python anzeigen wo man auf einer sehr hardwarefernen Abstraktion Befehle in den Computer eingeben kann. In dem Sinne dass sich fast als Pseudocode Algorithmen in einer Scriptsprache formulieren lassen und das man an Google anfragen senden kann wie "warum ist das Wasser nass?".

Ein Forth System kann man em ehesten als turing-mächtigen Computer auf der Stufe eines Kellerautomaten bezeichnen. Laut Wikipedia besteht ein Kellerautomat aus 2 Stacks und sieht der Abbildung entsprechend einer Turing-Machine ähnlich. Nur dass sich neben dem Schreiblesekopf noch eine Erweiterung befindet namens Stack.

Minimal instruction set computer Üblicherweise gibt es in der Informatik einen Konsens darüber ab wann ein Machinenmodell Sinn macht. Man unterteilt Computer in Lerncomputer welche theoretisch untersucht werden und praktisch relevante Machinen welche programmiert werden können. So ist zwar bekannt dass State-Machinen echte Computer in dem Sinne dass sie Schleifen haben und if-then-Befehle verarbeiten es ist jedoch nicht möglich damit etwas sinnvolles anzufangen, also Betriebssysteme für diese Computer zu entwickeln. Registermachinen gelten sowohl theoretisch als auch praktisch als sinnvoll, man kann sie wirklich programmieren. Und hier kommt Forth ins Spiel. Aus Sicht der Mainstream Informatik handelt es sich um Minimal instruction set computer die keine praktische Bedeutung besitzen. Man nimmt an, dass man sie nicht programmieren kann, jedenfalls nicht ökonomisch sinnvoll. Die Forth Community behauptet hingegen, dass es sehr wohl geht und hat dafür eine eigene Sprache namens Forth entwickelt. Die Frage lautet: wer hat Recht? Ist Forth eine rein akademische Sprache mit der man die Funktionsweise von Kellerautomaten erläutert oder kann man damit echte Probleme lösen?

Auf github gibt es ein Projekt namens "movfuscator" [16] dass C-Quelltext so aufbereitet, dass es auf einem Computer läuft, was nur den Mov Befehl kennt. Verwendet wurde dabei der LCC Compiler. Das neue Assembler Programm (mit den Mov-Befehlen) wird dann auf einer x86 Archtektur ausgeführt.

Obwohl es nicht als Forth Projekt gestartet ist, geht es ungefähr in diese Richtung. Die Annahme lautet, dass die Hardware ein Minimal Instruction Set Computer ist. Also ein System was nach heutigen Maßstäben als nicht-programmierbar gilt, es sei denn jemand ist so schlau einen C-Compiler dafür zu schreiben. Genau das ist offenbar möglich. Dadurch kann man selbst diese minimale CPU so programmieren wie eine normale x86 Architektur.

Verallgemeinert man die Lösung kann man sagen, dass die C-Programmiersprache den Unterschied definiert zwischen einem theoretischen und einem realen Computer. Wenn es für eine CPU einen C-Compiler gibt der dafür Machinencode erzeugen kann ist es möglich diese CPU auf High-Level-Ebene zu programmieren. Also all die Spiele und Primzahlensuchalgorithmen die es schon gibt darauf auszuführen. Wenn es hingegen nicht gelingt, dann gilt diese CPU als nicht praktisch relevant. Man kann sie zwar lowlevel programmieren aber es ist sehr aufwendig.

10 Ökonomische Aspekte

10.1 Einleitung

Warum sich Forth nie so recht hat durchsetzen können hat damit zu tun, dass die dahinterstehende Technologie eigentlich niemand benötigt.Will man Forth auf seine Quintessenz zusammendampfen ist es nichts anderes als eine Virtuelle Maschine für einen Stackcomputer. Sie lässt sich in 10 kb C Code programmieren. Oder noch

[16] https://github.com/xoreaxeaxeax/movfuscator

besser, man schreibt Forth in Forth selbst, so ähnlich wie Pypy nur mächtiger. Die Frage ist, was soll das? Wo ist der Nutzen? An sich besitzt Forth nicht mehr nutzen, als wenn man sich mit dem Busy-Beaver Wettbewerb oder mit Brainfuck die Zeit vertreibt. Es ist eine intellektuelle Übung ohne große Bedeutung. Warum Forth aber dennoch einen Wert besitzt kann man sehen, wenn man den Blick auf die ökonomischen Elemente der Computerhardware richtet. Es gibt hier Großkonzerne wie Intel und co, welche Unsummen mit ihren CPUs verdienen. Würde man das auf Forth als Standard-Sprache und Forth-Chips als Standard-Mikroprozessor umstellen ließe sich volkswirtschaftlich gesehen sehr viel Geld sparen. Die Prozessoren werden preiswerter, verbrauchen weniger Strom und leisten mehr.

Warum sich Forth in einer Nische der Informatik befindet ist simpel: es ist so gewollt. Und zwar von den Neoludditen, welche in technischem Forthschritt ein Problem sehen. Forth ist für heutige Computerhardware, dasselbe wie ein Benzin-Auto für die Postkutsche war: eine existenzielle Bedrohung.

Ein wenig naiv ist die Sache simpel: man fertigt Stackmachinen in Hardware, programmiert dafür neuartige Compiler und portiert die vorhandene Software darauf. Das ganze ist ein typisches Informatik-Thema. Dennoch ist der Umstieg auf Forth alles andere als alltäglich, der Grund ist dass Forth anders als die übliche Hardwarebranche einen massiven Technologiesprung bedeutet, in dem Sinne dass wie aus dem Nichts, die Computer um ein mehrfaches leistungsfähiger werden. Und diese Verbesserung will wohl dosiert sein. Es geht hier weniger um technische als vielmehr um kulturelle Fragen. Genauso gesagt um einen Kulturschock.

Obwohl Forth einen großen Vorteil verspricht verläuft die Einführung relativ bedächtig. Obwohl der GA144 Chipsatz fast zum Selbstkostenpreis verkauft wird, sind die Einstiegskosten in Forth beträchtlich. Das Problem ist aus der Linux Welt bekannt. Rein formal kann man die neueste Fedora Distribution kostenlos herunterladen und erhält damit das fortschrittlichste Betriebssystem. Nur, dennoch ist Linux keineswegs kostenlos, um es zu benutzen muss man sich erst einlesen. Und genau diese Hürde ist bei Forth um einiges höher. Genauer gesagt haben wir derzeit die paradoxe Situation, dass ein GA144 Chip scheinbar aus der Zukunft stammt, nur 20$ kostet, als Transputer ausgelegt ist und unglaublich mächtig ist, gleichzeitig aber die Anzahl der Personen die ihn korrekt oder auch weniger korrekt inbetriebnehmen können verschwindend gering ist. Das ist nicht so ein Problem von Forth, sondern es ist ein Bildungsproblem. Das heißt, es gibt keine guten Bücher dazu, und vor allem keine leichtverständlichen. Diejenige Literatur über Forth die es gibt, ist nicht zu gebrauchen, bzw. sie muss erst noch geschrieben werden.

Die Herausforderung besteht darin, ein komplexes Thema wie Forth für den Mainstream aufzubereiten um so den Kreis der User zu erhöhen. Wie das gehen soll ist nicht ganz klar, es gab zwar einige Versuche in der Vergangenheit aber erfolgreich waren die nicht. Das Problem mit Forth ist, dass diese Programmiersprache nicht im Jahr 2017 erfunden wurde und jetzt ähnlich wie ein Consumerprodukt in den Markt gedrückt wird, sondern Forth war schon immer da. Der Witz ist, dass die Literatur über Forth die 30 Jahre und älter ist, keineswegs veraltet ist, sondern technisch immernoch weit in die Zukunft reicht. Nicht Forth ist das Problem, sondern die Welt außerhalb von Forth. In enem Vortrag wurde ein Forth Entwicklerboard einmal als der teuerste Computer aller Zeiten bezeichnet. Und ja, das ist korrekt. Nicht rein formal, weil das Board für unter 100US$ verkauft wird, sondern teuer im Sinne dass man sich das ganze erst erarbeiten muss. Das heißt, man erwirbt das Entwicklerboard und ist damit dann die nächsten 50 Jahre erstmal ausgelastet.

10.2 Kosten von Vintage Computern

Warum historische Computer bis heute faszinieren ist simpel: sie sind extrem teuer. Jetzt mag man sich fragen, was an einem ausrangierten 1541 Diskettenlaufwerk der Marke Commodore den Preis großartig hochtreibt, weil auf die Floppydisk ja nur 170 kb draufpassen, aber genau das ist der entscheidene Punkt. Um diese Hardware sinnvoll einzusetzen muss man aus heutiger Sicht sehr viel darumherumbasteln. Theoretisch kann man heute immernoch auf einem C-64 Texte schreiben und im Internet surfen. Aber vorher muss man erstmal die benötigte Software komplett neu programmieren. Standard-Software läuft da nicht drauf.

Innerhalb der Vintage Computing Szene besitzen Stackmachinen einen besonderen Reiz. Sie waren schon zu Zeiten ihrer Einführung eine Absonderlichkeit, auch in den 1980'er gab es für den Novix Microcontroller kein Softwareangebot. Stackcomputer sind also nochmals eine Steigerungsform in Sachen Kostentreiber. Will man sie heute in Betrieb nehmen wird es richtig teuer. Technisch gesehen wäre es möglich, aber wer hat die benötigten Ressourcen? Das Spielen mit der idee ob es nicht doch gelänge aus einem Forth-Chips die letzten Resserven herauszulocken ist genau das, was bis heute an dieser Technologie fasziniert.

Mit diesem Hintergrundwissen kann man leicht definieren was Fortschritt bedeutet. Es geht darum die Kosten zu senken. Damit sind die Kosten für Hardware, Software und Projektkosten gemeint, also wie aufwendig es ist ein bestimmtes Ziel zu erreichen. Wieviel man erklären muss bis ein Außenstehender es verstanden hat. Das interessante an Vintage Computern ist, dass sie in ihrer jeweiligen Zeit als fortschrittlich wahrgenommen wurden. Der C-64 war einer Lochkartenbuchungsmachine haushoch überlegen. Es gab Standardprogramme mit denen man sofort losarbeiten konnte. Aus heutiger Sicht jedoch ist der C-64 sehr teuer. Obwohl er nur 64 kb Arbeitsspeicher besitzt ist seine Inbetriebnahme aufwendig und die Programmierung auch. Anders formuliert, ein Topmoderner Intel Xeon Server für 10000 US$ ist um einiges preiswerter als wenn man versucht dasselbe mit einem Commodore 64 zu erreichen. Obwohl Technologien wie Java, 14 nm Fertigung und Linux extrem kompliziert klingen sind sie doch einfacher als eine MOS 6502 CPU.

11 Abschnitte

11.1 Forth ist kein Joke

Neueinsteiger unterschätzen Forth. Sie glauben, es wäre keine richtige Sprache sondern nur ein Witz. Wo sich also Leute darüber lustig machen, wie jemand Forth nicht versteht um dann in einem Forum um Hilfe zu fragen. Generell kann man jedoch sagen, dass es außerhalb von Forth mehrere Esoterische Programmiersprachen wie Joy, Brainfuck und Haskel gibt die nicht ernstgemeint sind und auch nicht für den praktischen Einsatz konzipiert wurden. Üblicherweise zeichnen sich Joke-Programmiersprachen dadurch aus, dass sie concative funktioneren so wie der HP RPN Taschenrechner auch, und das sie stackbasiert sind. Wenn man eine neue Sprache entdeckt, die ebenfalls nach der umgekehrten polnishen Notation funktioniert und auf einem Stack aufbaut kann man fast sicher sein, dass es sich dabei um eine Kunstsprache handelt. Die einzige Ausnahme ist Forth. Diese Sprache ist ernstgemeint, die darin erstellten Programme erfüllen einen Zweck und das Ziel ist es, in Forth sinnvolle Betriebssysteme wie z.B. bigforth zu programmieren. Auch die Forth Chips wie der GA144 sind für seriöse Anwendungen wie neural modelling oder High-Performance-Computing entwickelt. Das es dort keine leistungsfähige RAM Speicher gibt hat nichts mit Forth zu tun, sondern liegt an den strikten Exportkontrollen der US-Industrie welche um ihr geistiges

Eigentum in Bezug auf Supercomputing fürchtet.

Was man jedoch feststellt, dass es innerhalb der Forth Community einige Komiker gibt, welche nicht wirklich an Forth interessiert sind sondern ihre eigene Ziele verfolgen. Diese haben es sich zur Lebensaufgabe gemacht, möglichst unsinnige Programme zu schreiben oder schwer verständliche Programme zu verfassen um sich dann darüber zu freuen, dass sie Neueinsteigern erklären können wie es funktioniert. Das ist jedoch die große Ausnahme. Forth an sich ist eine seriöse Sprache die positiv in die Gesellschaft wirkt und wodurch man etwas über Hardware und Software lernen kann.

Anders als Forth ist Pypy jedoch eine klassische Joke Programmiersprache. Diese ist stackorientiert, arbeitet mit der umgekehrten polnischen Notation und ist für echte Projekte nicht zu gebrauchen. Schon Python selber kann man nur als Witz bezeichnen, Pypy ist noch ein lächerlicher. Würde man die Energie welche in Python abfließt für Forth Projekte bündeln, hätten wir Weltfrieden.

Ein weiteres Witzprojekt was nicht taugt ist Micropython. Es handelt sich um eine Python Virtual Mascine in 200 kb welche auf Microcontrollern läuft und wo man ganz normal in Python programmieren kann, inklusive Vererbung. Das ganze ist jedoch nicht ernstgemeint, sondern Micropython ist lediglich eine Persiflage auf die mächtigen objektorientierten Features von Forth, wo bekanntlich mit mini-oof.fs eine Vererbung standardmäßig dazugehört.

11.2 Forth langsamer machen

Forth gilt als eine der effizientesten Möglichkeiten einen Computer zu programmieren. C kann man sehr gut mit Forth vergleichen, und zwar ist C bedeutend lansamer. Insbesondere wenn man den Vergleich auf einer dezizidierten Stackmachine ausführt. C ist nur eine Hochsprache, Forth hingegen agiert direkt auf Bare Metal. Aus Sicht von Forth ist damit alles gesagt, aber nicht so schnell. Ist Forth wirklich die Zukunft? Funktioniert Informatik tatsächlich so, dass es um Effizienz geht?

Schaut man sich einmal die Entwicklung an die mit UNIX angestoßen wurde, so hat sich C nicht deshalb durchgesetzt weil es unglaublich effizienten Code generiert und die Z80 CPU ist auch nicht deshalb zum meistverkauften Prozessor aller Zeiten geworden, weil sie schneller war als eine dezidizierte Forth CPU, sondern diese Dinge haben sich durchgesetzt weil sie schon bei ihrer Markteinführung hoffnungslos veraltet waren. Im Grunde sind Registermaschine komplett überholt. Es ist nicht möglich sie preiswert zu fertigen oder sie parallel zu schalten. Die Technologie hinter Registermaschinen basiert auf einem veralteten Konzept was den CPU Kern als eine Art von Hauptspeicher betrachtet in dem möglichst viele Register das Caching übernehmen. Und Compiler die auf dieser Architektur aufbauen machen es nur noch schlimmer. Diese Ineffizienzen sind jedoch genau das was man als Programmierer benötigt. Die Gewissheit, dass die Maschine der man gegenübersitzt nicht das beste Gerät ist was gebaut wurde,sondern aus einer Entwicklungslinie abstammt welche als Ausschuss bezeichnet wird.

Wer in C programmiert hat es mit einem dysfunktionalen System zu tun. Mit einem alten Röhrenradio was einen Wackelkontakt hat. Noch schiimmer als C ist höchstens noch Python. Nicht nur, dass das Pypy Projekt grandios damit gescheitert ist, die Performance zu steigern sondern auch die objektorientierten Features von Python sind fehlerhaft.

Aber auch hier wieder ist es genau das, was gewünscht ist. Die Leute lieben ihr Python, gerade weil es so unglaublich ineffizient ist, gerade weil es nicht möglich ist mit Python komplexere Programme zu schreiben. Was wir brauchen sind nicht etwa Effizienzsteigerungen sondern die Zukunft sieht so aus, Sprachen zu nutzen die noch viel nutzloser sind als Python und Computer zu bauen, die noch viel ineffzienter arbeiten als die heutigen Intel CPUs. Im Grunde ist es fast schon eine Frechheit, dass moderne CPUs keine Lüfter mehr benötigen. Das ist nicht der richtige Weg. Besser ist es, CPUs so zu übertakten, dass sie eine Wasserkühlung benötigen, weil diese Leckschlagen kann und man dann weitere Schwierigkeiten zu lösen hat.

Würde man die Hardware auf leistungsfähige Stackmachinen aufrüsten braucht man auch keine Kühlung. Die Systeme würde geräuschlos arbeiten und sie wären supereinfach zu programmieren. Will das jemand ernsthaft? Wohl kaum, die Welt wäre langweiliger wenn es zuviel Perfektion gibt.

11.3 Fixed Gear Fahrräder haben keine Bremsen

In der Welt der Informatik ist Forth einmalig. Keine andere Programmiersprache spaltet so sehr die Gemüter. Woran das liegt lässt sich leicht sagen. Bei Forth fehlt etwas. 1 auf Seite 3 war das Eingangsbesipiel mit dem “Hello World Sourcecode” in Forth. Wenn man sich den Code nochmal anschaut sind es die fehlenden Klammern, genauer gesagt die Parameterübergabe, die die Sache so luftig erscheinen lassen. Irgendwie ist das ungewöhnlich und cool gleichermaßen. Zumindest außerhalb der Informatik gibt es etwas vergleichbares. Sogenannte Fixed Gear Fahrräder, besser bekannt als Bahnrad, sind gleichermaßen wie faszierend wie gefährlich. Auch dort fehlt eine wichtige Eigenschaft: die Bremsen, es gibt übrigens auch keinen Freilauf, sondern das Rad ist direkt mit der Kette verbunden. Fixed Gear Fahrräder wurden erfunden um das Fahrradfahren lernen zu erleichtern. Genauer gesagt haben sie das Ziel eine einheitliche Trittfrequenz zu ermöglichen. Es wirkt, als wenn Mensch und Maschine miteinander verschmelzen. Ungefähr so kann man auch Forth einstufen. Forth wurde erfunden, um das Programmieren zu erlernen. Gleichzeitig sind Forth bzw. Fixies für den Straßenverkehr nicht zugelassen. Das ganze ist nicht nur ein Frage der Technik sondern aucn der Kultur. FIxies sind genauso wie Forth Teil der Counterculture. Also einer Strömung welche sich zu dem Thema als Pro positioniert und dadurch die Mainstream Gesellschaft ablehnt.

Warum Fixies für den Straßenverkehr nicht zugelassen sind ist simpel. Wenn man damit einen Berg herunterfährt und versucht zu skidden (also das Hinterrad ausbrechen lässt) besteht eine hohe Unfallwahrscheinlichkeit. Damit gefährdet man sich und andere Verkehrsteilnehmer. Warum Stackmachinen in der Mainstream-Literatur nicht oder sehr abfällig beurteilt werden hat ähnliche Gründe. Rein theoretisch kann man sie bauen, aber das gibt mehr Stress als es nützt.

Wie sieht dieser Stress konkret aus? Nun, wenn man Programmieren lernen möchte, geht es nicht nur um den Computer an sich, sondern auch um die Kultur dahinter. Das was der Mainstream als korrekten Umgang mit Computern einschätzt ist die UNIX Culture. wo man also mit einer Hochsprache wie C ein Betriebssystem schreibt und darauf aufbauend dann Erweiterungen geschrieben werden. Es geht darum, diese Kultur zu verinnerlichen. Wenn man hingegen Forth lernt, wird man Teil einer anderen Bewegung. Der Forth Kultur, diese ist inkompatibel zu UNIX wie auch C gleichermaßen. Und wo immer Forth Experten auf Unix Experten stoßen gibt es Streit.

Jetzt kann man sagen, hey Leute so schlimm ist das nicht, es sind doch nur Computer. Doch damit wird man der Sache nicht gerecht. Auch Fixies haben mit normalen Fahrrädern keine Gemeinsamkeit, vielmehr ist der Kontrast so gewollt. Forth ist nicht kompitbel mit C. Und wird es auch nie sein. Forth ist böse.

Um es klar zu sagen, Forth hat mit Informatik nicht viel zu tun. Es geht eher um eine Lebenseinstellung, also das Bejahen von etwas was wichtig ist. Forth grenzt sich nicht bewusst vom Mainstream ab, sondern es war schon immer anders.

Schauen wir nochmal auf das Fixed Gear Fahrrad. Kann man so

nicht herumfahren, wo ist das Problem? Das Problem ist, dass das Fahrrad keinen Freilauf besitzt, wenn man die Füße von den Pedalen nimmt, drehsn sie sich weiter. Das Problem mit Forth ist so ähnlich. Es hat keine Register. Dadurch ist es ncht möglich, effiziente Hochsprachencompiler dafür zu schreiben. Und ohne Compiler auch keine Betriebssysteme. Generell geht es also um die Frage, wieviel Minimalismus möglich ist.

Vielleicht dazu ein konkretes Beispiel: Angenommen man hat eine 8080 CPU vorliegen und möchte dafür einen C Compiler schreiben. Man wird feststellen, dass die CPU arg wenig Register besitzt und es deshalb nichts wird mit dem Compiler. Jetzt gibt es zwei Mögliche Antworten. Entweder erhöht man die Registerzahl um dann leichter einen C-Compiler programmieren zu können. Oder man lässt die Register so wie sie sind und geht sogar noch mehr in Richtung Minimal Instruction Set so dass man überhaupt keine Hochsprachencompiler mehr entwickeln kann. Letzteres wird als Forth-Style bezeichnet.

Selbst mit heutiger Technik ist es nicht möglich für einen Minimal Instruction Set Computer effiziente Hochsprachencompiler zu programmieren. Das passt irgendwie nicht zusammen. Der Mainstream hat auf das Problem derart reagiert, dass er Software wichtiger als Hardware bewertet. Das heißt, Intel baut einfach jene Prozessoren für die es C-Compiler gibt und macht genau das, was die Softwareentwickler wünschen. Während man bei Forth den umgekehrten Weg eingeschlagen hat und die Bedürfnisse von C Programmierern konsequent ignoriert und stattdessen ein Umdenken beim Erstellen der Programme fordert. Das man also in Konzepten einer Stackmachine denken soll und nicht wie bei C auf den Code fokussiert.

Gewonnen hat bei diesem Spiel Forth, natürlich. Forth steht heute besser da denn je. Die Community ist besser aufgestellt, die Prozessoren sind leistungsfähig und der Vorsprung gegenüber Intel gewaltig. Der Verlierer hingegen ist der Mainstream. Er kämpft mit hohen Fertiungskosten, Abwärme durch die Prozessoren und fehlenden Standards.

Ohne Betriebssystem Die wichtigste Eigenschaft von Forth-CPUs ist, dass es dafür kein Betriebssystem gibt. Das ist jedoch kein Mangel, sondern ist der Auslöser für verschiedene Workarounds. Die Kultur in der Forth Community basiert darauf, über Gegenstrategien zu beraten wie man auch ohne Hochsprachencompiler und ohne Betriebssystem auf einem Chip rechnen kann. Die Antworten darauf sind vielfältig. Mit einer Ausnahme. Die Antwortet lautet nicht, die Anzahl der CPU Register zu erhöhen, so dass man einen C-Compiler auf die Plattform portieren kann. Dann wäre es kein Forth System mehr sondern etwas anderes. Ungefähr so ähnlich ist auch die Mentalität auf einer Skidding Veranstaltung. Anstatt einfach das Fahrrad aufzubocken und den Freilauf nachzurüsten hat man sportliche Methoden entwickelt wie man das Fahrrad anders abbremst als es normalerweise üblich ist. Die interessantesten / gewagtesten Mannöver werden beklatscht. Wenn das Skidding jedoch mißlingt, schrammt man mit dem Ellenbogen auf dem Asphalt entlang. Wenn man es auf einem Forth Treffen nicht schafft ohne Betriebssystem auszukommen, passiert nach dem Einschalten gar nichts. Das heißt, die CPU bootet nicht, es ist kein Bild zu sehen. Wenn man n icht weiß wie das geht mit Forth kann man daran auch nichts ändern.

Die Fixie Biker spielen mit dem Risiko einen Unfall zu bauen, als Forth User spielt man damit den Computer nicht sinnvoll einsetzen zu können. Das wird komischerweise als Befreiung verstanden. Ja einige Zuschauer applaudieren sogar explizit bei solchen Fail-Situationen. Also wenn jemand einen Forth Chip mitbringt ohne die passende Software liefern zu können. “Bravo” heißt es dazu aus dem Publikum. Und “Zeig mal her, das will ich auch haben”.

Damenfahrräder Vielleicht sollte an dieser Stelle einmal beschrieben werden, wie ein bequemes und alltagstaugliches Fahrrad aussieht. Zunächst einmal besitzt es einen Freilauf, was bedeutet, dass man aufhören kann mit dem Treten und das Fahrrad weiterrollt. Weiterhin befindet sich am hinteren Rad ein Rockschutz wodurch man auch in Alltagskleidung fahren kann und ganz wichtig, es gibt eine Vorder- und eine Hinterbremse, so dass man auch ohne einen Skid-Stop anhalten kann. Zu guter letzt sind helle Beleuchtungen vorne und hinten ein sinnvolles Accessoire sowie ein Korb in dem man Lebensmittel verstauen kann. Auch ein Kettenschutz, ein Fahrradständer sowie eine am Lenker angebrachte Klingel sind empfehlenswert. So und nicht anders sieht ein Straßentaugliches Fahrrad aus, mit dem man gemütlich und unter Beachtung der Verkehrsegeln über die Fahrradwege von New York City fahren kann. An einer roten Ampel hält man vorschriftsgemäß an und wenn man die Seite wechselt zeigt man dies an, damit der nachfolgende Verkehr sich darauf einstellen kann.

11.4 Forth bloßstellen

Eigentlich ist Forth eine sehr schöne und mächtige Programmiersprache. Wenn man sich für Computersprachen interessiert wird man vielleicht zur der Erkenntnis gelangen, dass Forth die beste Sprache ist die jemals erfunden wurde. Insbesondere wenn man in Forth einen rekursiven Algorithmus aufstellt der eine Turtle Grafik zeichnet wird man von seiner Einfachheit und Eleganz beeindruckt sein. Bei Forth gibt es aber auch etwas, was nicht so gut funktioniert. Und zwar das Veranstalten von Wettbewerben.

Eigentlich ist das etwas was alltäglich ist in der Informatik. Man denkt sich eine Aufgabe aus, und verschiedene Teams müssen dann in vorgegebener Zeit eine Lösung finden, am Ende wird ein Sieger gekührt. Es gibt solche Wettbewerbe um in einer 1 Stunde computerspiele zu programmieren oder um Roboter auf einer Linie langfahren zu lassen. Komischerweise ist sind die Hardcoresprachen Forth, APL und Lisp jedoch weitestgehend Challange-Freie Zonen. Es gibt keine oder nur sehr zaghafte Bestrebungen in diese Richtung.

Wenn man einen typischen Vortrag zu Forth lauscht wird man irgendwann zu der zynischen Erkenntnis gelangen, dass Forth eine sinnlose Sprache ist. Ich meine, wozu braucht man ein Word. Häufig wird dann der Sinn damit erklärt, dass man damit weitere Wörter definieren kann, doch das ist eine Erklärung innerhalb des Forth Ideologie. Und schnell stellt sich bei Einsteigern der Verdacht auf, dass sie niemals werden Forth verstehen. Mit Forth hat das jedoch nichts zu tun, sondern mit einer Unterrichtssituation ohne Fokussierung auf Anwendung. Generell besteht guter Unterricht aus zwei Teilen: einmal die vermittlung von theoretischen Grundlagen, ohne die es nicht funktioniert und anschließend dann der praktische Teil wo es um die Anwendung geht. Bei Forth wird meistens auf den 2. Teil verzichtet. Das ist schade, aber man kann es ändern.

Wenn man Forth, APL oder Haskel in einer Challange einsetzt, erweitert man die Aufgabenstellung. Es geht nicht länger um Wissen was der Lehrer hat und wo man selber nur zuhören kann, sondern in einer Challange geht es gar nicht primär um die Sprache sondern um ein soziales Spiel. Wer schafft es, das beste Programm zu schreiben, wer hat die kreativste Lösung.

Bei dezidzierten APL und Forth Wettbewerben kann man einen interessante Aspekt beobachten. Und zwar verschieben sich die Problemlösungsstrategien in Richtung Mainstream. Normalerweise wird APL in den Lehrbüchern als Listenverarbeitende funktionale Sprache mit einer sehr sondernbaren Syntax dargestellt. Will man jedoch mit APL ein Spiel unter Wettbewerbssituationen programmieren wird im Regelfall nicht das reine APL eingesetzt, sondern eines was um objektorientierte Features erweitert wurde. Das heißt, APL wird zwangsläu-

fig in Richtung Java umfunktioniert. Und exakt das passiert auch mit Forth. Will man mit Forth in einem Wettbewerb gewinnen, wird man ebenfalls vorher den Interpreter um OOP Features aufrüsten und einen programmierstil verwenden der eher nach C++ als nach Forth aussieht. Und zwar, weil die theoretischen und praktischen Anforderungen sich unterscheiden.

Versuchen wir einmal etwas näher einzutauchen in “Forth programming challanges”. Es ist erstaunlich schwer konkrete Beispiele zu finden. Als erstes gefunden habe ich den “Forth Benchmark Wettbewerb”. Es handelt sich dabei um ein Forth Programm, was man auf unterscheidlichen Vintage Computern wie Atari ST und C-64 laufen lässt und was die Performance ermittelt. Aber, das ist kein Progrmamierwettbewerb, sondern die Challange wird zwischen den Computern ausgetragen. Dort gewinnt beispielsweise der C-64 weil er am schnellsten die Primzahlen ermittelt hat. Ein Programmierwettbewerb wird jedoch zwischen Menschen veranstaltat, wo man also 4 Stunden Zeit hat um Code zu schreiben.

Im Usenet unter “comp.lang.forth” gibt es ebenfalls keine Berichte über durchgeführte Forth Challanges. Das einzige was ungefähr in diese Richtung geht, war eine Diskussion über den Forth Day 2003, dort wurde abschweifend diskutiert, ob man nicht eine programmier Challange durchführen sollte. Das wurde abgelehnt:

> “Now this kind of real world programming challange would emphasis just how different this kind of programming is from the sort of programming that many people do regularly. Suddenly a trivial Hello World program does not look quite so trivial. ”[17]

Hat die Forth Community womöglich Angst vor einer Programmierchallange? Schaut man sich einmal an, wie der Forth Day 2003 und alle nachfolgenden Treffen durchgeführt werden, so gibt es dort Präsentationen, leckere Verpflegung, Microcontroller zum Anfassen und sehr neue LED Beamer, aber es gibt keine Programming Challanges. Das ist doch ein wenig merkwürdig oder? Sollte ausgerechnet die mächtige Sprache Forth ungeeignet sein um darin Programmierwettbewerbe zu veranstalten? Wohl eher nicht. Forth ist turingmächtig und man kann darin ausgezeichnet programmieren.

Der Begriff “Programming Challange” wird in der Forth Community in einem semantisch anderen Kontext verwendet. Als Challange wird üblicherweise in einem Text eine Herausforderung bezeichnet. So wird in der Einleitung ein Primzahlalgorithmus als Challange bezeichnet um dann im weiteren Verlauf das dazugehörige Forth Programm zu entwickeln. Soweit ist das auch in Ordnung, aber Challange heißt mehr als einfach nur ein Beispiel zu bringen, eine Challange ist ein sportlicher Wettbewerb der wirklich durchgeführt wird und nicht nur eine literarische Figur um einen didaktischen Text aufzulockern.

C Programming Challange Technisch gesehen geben ich der Forth Community recht, wenn sie C als schlechte Programmiersprache bezeichnet. Es ist nicht besonders schwer darin zu programmieren, Forth ist weitaus interessanter und tiefgründiger. Aber, rund um die C-Community haben sich Programmierwettbewerbe etablieren können, diese sind fast noch wichtiger als die eigentliche Sprache. Diese Wettbewerbe dienen dazu, in einen sportlichen Wettstreit zu treten und verlieren zu lernen. Das fehlt ein wenig in der Forth Community. Die Technologie die sie verwenden ist besser, aber es fehlt die soziale Einbettung. Ja fast hat man den Eindruck, als ob das so gewollt wäre, dass man mit forth nichts sinnvolles anfangen kann und vor allem dass man in einem Programmierwettbewerb keine Preise gewinnen kann.

Warum man für die Sprache C Programming Challanges veranstaltet ist simpel: weil die Sprache selber nichts zu bieten hat. Sie benötigt anders als Forth sehr viel Speicher, lässt sich nicht effizient threaden und rekursion ist verpönt. Insofern dienen Wettbewerbe dazu, dieses Manko auszugleichen.

Besonders in Europa sind Programmierwettbewerbe in Anfängersprachen wie Java und C sehr beliebt. Weil man gegenüber den USA sehr viel aufzuholen hat und das eine Möglichkeit ist, das auf unterhaltsame Weise zu tun.

Micromouse Mit etwas suchen habe ich doch noch einen dezidierten Forth Programmierwettbewerb gefunden. Jedenfalls fast. In der Micromouse Challange müssen winzig kleine Microcontroller programmiert werden, auf denen üblicherweise Forth vorinstalliert ist. Auch die Vierte Dimension berichtet regelämäßig über diesen Wettbewerb. Micromouse ist eine der seltenen Gelegenheiten wo die Programmiersprache Forth unter Wettbewerbsbedinungen verwendet wird.

Aber, Micromouse ist kein dezidierter Forth Wettbewerb, und trotz intensiver Google Suche habe ich kein einziges Sourcecodebeispiel finden können. Üblicherweise wird dort C oder Assembler eingesetzt um die Roboter ins Ziel zu bringen. micromouse "forth programming" Von einem Teilnehmer Duncan Louttit gab es wohl mal einen Vortrag auf einer Forth Konferenz, den Sourcecode zu seinem Maze solver will er uns aber nicht verraten. Es ist also tasächlich so, dass Forth und Coding-Challanges nicht zueinander finden.

Das ist inhaltlich technisch sehr ungewöhnlich, weil Forth sich eigentlich ausgezeichnet eignet um Microcontroller zu programmieren. Und auf Forth Tagungen sind Microcontroller regelmäßig ein Thema. Und mehr noch, es gibt in der Forth Community auch sehr viele Paper die sich mit Robotics beschäftigen. Ich glaube, das Problem was besteht ist nicht so sehr die technische Seite, sondern dass die Forth Leute irgendwie zu Programmierwettbewerben auf Abstand gehen. Würde es nicht um einen Roboterwettbewerb sondern um eine Coding-Challange für Natural Language gehen, würde sich Forth vermutlich ebenfalls sehr zurückhaltend zeigen, obwohl auch hier eine lange Tradition in Bezug auf Forth und Natural Language Processing besteht.

11.5 Einsatzszenarien von Forth

Forth hat als Vorteil gegenüber Linux sehr viel minimaliischer zu sein und als Nachteil, dass es schwer ist in Forth zu programmieren. Den Minimalismus von Forth kann man produktiv einsetzen, wenn man sehr kleine Microcontroller programmieren möchte. Also Systeme wo kein Linux Kernel draufpasst. Aber Forth eignet sich auch gut, wenn man organische Computer programmieren will. Nanoroboter die aus molekülen gefertigt sind, bringen ebenfalls nur wenig CPU Leistung und wenig RAM auf die Wage. Theoretisch kann man darauf auch die Programmiersprache Brainfuck ausführen, Brainfuck hat jedoch den Nachteil ein wenig zu minimalistisch zu sein. In Brainfuck selber Code zu schreiben ist schwer. Langjährige C Programmierer werden einwenden dass man ja auch C einsetzen kann und über einen compiler daraus Assembler Code erzeugen könne. Nur, erstens benötigt man dafür einen Crossassembler und zweitens ist der generierte Assemblercode nicht wirklich minimalistisch. Wenn genuügend CPU Leistung bereitsteht wie auf einem Raspberry PI Computer ist das kein Problem, aber wenn nicht bleibt eigentlich nur Forth als Möglichkeit.

Kommen wir zu einem Grund der gegen Forth spricht. Und zwar gilt die Programmierung als schwierig. Es ist schwerer in Forth ein Programm zu erstellen als in C und vor allem gibt es für Forth keine gute Bibliotheken. Will man etwas programmieren muss man das Rad nochmal neu erfinden. Desweiteren ist der Umgang mit dem Stack kompliziert, bis man eine simplen Primzahlalgorithmus in Forth fertiggestallt hat vergehen viele Stunden. Aber, in der Praxis ist der

[17] https://groups.google.com/forum/#!searchin/comp.lang.forth/challange|sort:relevance/comp.lang.forth/2n8Fa89k-OM/pqr5d8bDVFsJ

Aufwand vernachlässigbar. Und zwar weil man Forth nicht verwendet um darin Algoirthmen zu erfinden oder sich auszudenken wie man generell einen Roboter programmiert sondern Forth ist eine Zielsprache wenn man den Algorithmus bereits in Python vorliegen hat und ihn lediglich auf einem Microcontroller ausführen möchte. Ein idealer Workflow mit Forth-in-the-loop sieht so aus, dass man als Python Programm einen leistungsfähigen RRT Pfadplanner programmiert und optimiert hat, diesen ausdruckt und den Code manuell für Forth umschreibt.

Forth steht selbstverständlich in Konkurrenz zu anderen Programmiersprachen. Ein heißer Anwärter Forth vom Socken zu stoßen ist das erwähnte Python. Mit Micropython gibt es eine Distribution welche auf Microcontrollern läuft, wo man also 1:1 den Code der auf dem Desktop läuft übertragen kann und überhaupt nichts mehr umschreiben muss. Aber, Micropyhton stellt im Vergleich zu Forth relativ große Anforderungen an die CPU. Weiterhin ist von Python bekannt dass die Echtzeitfähigketten und Multitasking-Eigenschaften nicht die besten sind. Und minimalistisch ist Python ganz sicher nicht. Es ist vergleichbar, als würde man C# auf einem Microcontroller ausführen. Es geht zwar auch, aber es ist nicht sehr nahe an der Hardware.

Ein weiterer Vorteil von Forth ist, dass man kein seperates Betriebssystem benötigt. Forth bringt sein eigenes BIOS mit, seinen eigenen Kernel und seinen eigenen Editor. Eine Sprache die noch schlanker ist, gibt es nicht. Forth ist gewissermaßen die unterste Ebene.

11.6 Auf Forth verzichten

Forth definiert sich selbst als minimalistische Programmierumgebung. Und bis heute kann kein C-Compiler oder linux-artiges Betriebssystem es in dieser Hinsicht mit Forth aufnehmen. Aber ist deshalb Forth unersetzlich? Wahrer Minimalismus beginnt dort wo man auf Forth verzichtet, also auf einen Minimalismus nicht angewiesen ist und sich stattdessen einen 32 bit ARM Microcontroller kauft, dort Micropython installiert und dann schön ineffizient seinen Micromouse Roboter ins Ziel bringt. Der Clou ist dass der Verzicht auf Forth keine Nachteile mit sich bringt. Dadurch wird der Roboter keineswegs langsamer. Der eigenliche Flaschenhals bei Micromouse ist nicht die Frage ob nun 10 kb oder 2000 kb an RAM Benötigt werden sondern der Flaschenhals ist eher in der High-Level-Programmierung zu suchen, also welche Algorithmen verwendet wurden.

Richtig ist, dass wenn man Forth verwendet man damit seine schöne Roboter bauen kann. Aber Forth ist keine Grundfertigkeit die man haben muss weil alternativ das komplette Projekt scheitert. Sondern Forth ist eher ein luxoriöses Addon. Hat man kein Forth ist das kein Nachteil. Was jedoch ein Nachteil ist und hier beginnt der wirkliche Minimalismus, ist wenn man keine Ahnung von UML Diagrammen und Objektorientierter Programmierung in Python besitzt. Weil ohne das, man kaum wird eine leistungsfähige Robot-Control-Software wird entwickeln können. Nur mit Forth oder C ist es nicht möglich sowas zu tun.

Minimalismus ist laut Definition das Weglassen von überflüssigem Beiwerk. Objektorientierte UML Diagramme kann man nicht weglassen, und das git Versionsverwaltungssystem auch nicht, weil sonst das komplette Projekt scheitern würde. Worauf man jedoch verzichten kann ist es, sich in Forth einzuarbeiten und die Programmierung eines Stacks zu verstehen. Diese Fähigkeiten sind entbehrlich, es gibt gute Alternativen dazu.

Es mag ein wenig sonderbar klingen wenn man eine minimalistische Forth Umgebung als Luxus bezeichnet, aber so ist es. Den Minimalismus von Forth muss man sich leisten können. Genauer gesagt geht es darum, Dinge zu verwenden die durch etwas anderes ersetzt werden können ohne die Funktion zu beeinträchtigen. Einen Forth Microcontroller kann man leicht gegen ein C-Microcontroller oder ein Python Äquivalent austauschen. Und ein Stackprozessor lässt sich durch größere Modelle die mehr Geld kosten ersetzen. Die Frage ist natürlich warum man das tun sollte. Der einfachste Fall ist einfach Zufall. Das heißt, wenn irgendwann der Forth Microcontroller kaputt geht und kein anderer da ist, dann nimmt man eben einen wo Linux vorinstalliert ist. Das ist zwar nicht so klein und schlank aber zur Not geht es auch. Ich will nicht behaupten es gerne zu sehen, wenn Forth ins Abseits gedrängt wird, das Problem ist nur dass Forth keine Qualitäten mitbringt die nicht auch andere besitzen.

Rein vom technischen her stellt Forth das Optimum in Sachen Microcontrollerprogrammierung. Kein anderes System ist so stromsparend so leicht zu installieren und so effizient wie ein eForth System. Aber man sollte nicht vergessen, dass Microcontroller nur Mittel zum Zweck sind, sie dienen dazu eine Aufgabe zu erledigen und man ist nicht abhängig von Forth. Wovon man hingegen abhängig ist, dass ist Softwareentwicklung und zwar insbesondere solche bei denen größere Projekte das Ziel sind. Wer nicht in der Lage ist komplexe Systeme zu entwerfen und zu debuggen, dem nützt der beste Microcontroller gar nichts. Er kann zwar hübsch die Lampen blinken lassen, aber das reicht nicht aus.

11.7 Python vs. Forth

Warum die Leute Python einsetzen und nicht C ist simpel. Zwar ist die Sprache C besser auf Microcontroller abgestimmt aber es ist aufwendig, in C ein WLAN Modul anzusteuern, einen Webserver aufzusetzen oder Live-Daten zu visualisieren. Anders gesagt, Microphyton ist das worauf die Welt gewartet hat. Und noch mehr, genauso gibt es zahlreiche Gründe warum die Leute lieber Python anstatt mit Forth ihr Glück zu versuchen. Es sind die selben Gründe und noch einige mehr. Ein Forth System lässt sich zwar auf jeden Microcontroller installieren aber etwas damit anfangen kann man nicht. Will man einen Pfadplanner programmieren muss man sich erstmal in die Arrayprogramming in Forth vertiefen. Da ist aufwendig. Will man einen Webserver aufsetzen muss man ebenfalls manuell programmieren. Hat man hingegen Micropython reicht ein simpler Befehl und der Server ist online. Genau genommen können sich Python User also den Minimalismus von Forth nicht leisten. Der Witz ist, dass ein größerer Microcontroller mit mehr Speicher effektiv billiger kommt als wenn man einen Forth Microcontroller verwendet. Warum? Weil bei Micropython alles out-of-the-box mit dabei ist. Man kann damit zügiger ein Projekt durchführen und kommt schneller zu Resultaten.

Wenn man von Forth auf C wechselt ist das für einige Offenbarhung. Plötzlich kann man in der Hälfte der Zeit die doppelte Menge an Code schreiben. Man muss sich nicht zweimal überlegen wie das mit dem Stack geht sondern schreibt einfach hin was man machen will. Noch viel angenehmer ist der Umstieg von C auf Python. Dort ist die Programmierung noch leichter, und das Debuggen geht noch schneller.

11.8 Transcompiler, Konverter und Compiler zu Forth

Das man Forth nativ programmieren kann ist bekannt. Leider ist das Erstellen von dezidiertem Forth Code sehr schwer. Umso nützlicher wenn man Konverter hat die in die Forth Sprache automatisch übersetzen. Die Auswahl solcher Konverter ist nicht sehr hoch, aber es gibt sie:

- C to Forth: https://github.com/dmedinag/C-to-Forth-compiler basiert auf einer Bison-Datei um den Parsergenerator zu erzeugen. Damit kann dann eine “.c” Datei in eine “.fs” Datei konvertiert werden.

- Brainfuck to forth: https://github.com/kmyk/forth-to-brainfuck ausprobiert habe ich diesen Konverter noch nicht. Laut Beshreibung soll er die esoterische Programmiersprache Brainfuck in Forth Code übersetzen können.

- Scheme to Forth: http://www.complang.tuwien.ac.at/schani/oldstuff/#schemeinforth Scheme ist ein Lisp Dialekt der ähnlich wie Forth einen Metaprogrammieransatz verfolgt. Leider ist unklar wie das Programm aufgerufen wird, ich konnte es nicht testen.

11.9 Ein Betriebssystem für Forth

Eine Programmiersprache wie Forth ist sinnlos wenn man kein Betriebssystem dazu hat. Leider ist zu diesem Thema die vorhandene Literatur sehr dürftig. Genannt werden folgende Betriebssysteme die in Forth geschrieben sind:

- ForthOS (unabhängige Entwicklung, komplett ohne Dokumentation)

- 4os (war auf dem iTV Forth PC vorinstalliert)

- bigforth (Anleitung auf Deutsch vorhanden, wurde ursprünglich für den Atari ST entwickelt und besitzt ein objekorientiertes Fenstersystem)

Nach etwas Suchen bin ich noch auf einen Begriff gestoßen, der relevant sein dürfte: OSkit. Ausgeschrieben heißt es “Flux OS Toolkit” [4]. Gemeint ist eine Sammlung von sehr allgemein gehaltenen Routinen mit denen man ein Betriebssystem automatisch generieren kann. In welcher Programmiersprache OSkit entwickelt wird ist unklar. In der Theorie kann man damit auf Knopfdruck ein beliebiges Betriebssystem erzeugen und spart sich das was Linus Torvalds als Kernel Development bezeichnet.

Die Vermutung liegt nahe, dass auch ForthOS und 4os mithilfe von OSkit erzeugt wurden. Also keineswegs da jemand 50 MB an Sourcecode in der Sprache Forth eingetippt und gedebuggt hat, sondern die Software aus dem Generator kam. Die Idee geht ungefähr so: Zuerst portiert man ein Forth System auf eine neue Plattform, wie z.B. x86. Dann nimmt man eine kleine Zusatzdatei die nicht größer ist als 2 kb um aus dem Forth System einen Scheme Interpreter zu machen. Dann lasst man darin das OSkit laufen und erzeugt den Code für einen Kernel, dann wird auf ähnliche Weise ein Lisp kompatibler Window Manager erzeugt und so kann man das komplette Betriebssystem inkl. Spiele und TCP/IP aus sehr wenigen zeilen Code erzeugen. Metaprogramming heißt das in der Fachsprache. Also Programmieren ohne zu programmieren. Ob sich das ganze wirklich realisieren lässt ist unklar, reine Spekualtion das ganze.

11.10 Schriftkultur

In einer Schriftkultur werden Informationen in Büchern notiert. Mit den Informationen hat das zunächst nichts zu tun, sondern mit dessen gesellschaftlicher Bedeutung. Man kann auch in einer nichtschriftlichen Gesellschaft ein hervorragender Arzt sein, das Wissen über Anatomie ist das selbe. Der Fokus liegt also auf Kultur, und meint wie Arbeit geteilt wird und wie Wissen organisiert wird. Schauen wir uns jetzt einmal die Forth Communit und ihrem Streben nach Sourceless Development an. Es bedeutet dass man auf eine Schriftkultur verzichtet. Genauer gesagt verzichtet man auf das Erstellen von Libraries. Über das Programieren oder Informatik sagt das allein nichts aus, sondern Forth kann man eher als Gesellschaftsbild bezeichnen. Also wie die Arbeit verteilt wird und der Zugang zu Informationen geregelt ist. Wenn man keine Bücher und keine formal niedergeschriebenen Betriebssystembibliotheken nutzt wird die Ausbreitung von Wissen verhindert. Es entsteht eine stark hierarchiche Gesellschaft bei der einige wenige über die Interna eines Computers informiert sind und dieses Wissen für sich behalten. Das ist die Essenz von Forth. Sage nichts, schreibe nichts auf, behalte dein Wissen für dich. So lautet das Mantra. Wo man keinen Sourcecode als Bibliothek aufschreibt und wo man keine Bücher über Compilerbau publiziert gibt es auch keine Gegenseite die diese lesen kann. Man bleibt unter sich, schottet sich ab, grenzt sich ab.Forth mit dem Mittelalter zu vergleichen ist so falsch nicht. Auch im Mittelalter war das Wissen auf die Kaste der Mönche beschränkt, sie waren die einzigen welche Latein konnten. Außenstehende erhielten keinen Zugang. Technisch gesehen mag das funktionieren. Im Mittelalter hat man sehr gutes Latein gesprochen. Ob Wissensmonopole jedoch gut sind wenn man Mitbestimmung, Aufklärung und freies Denken fördern möchte ist fraglich.

Forth funktioniert auf eine simple Weise. Man verteuert die Ware Information und erschwert dadurch den Zugang. Wenn es keine Betriebssysteme gibt, kann man auch nicht davon lernen, wenn es keine Bücher gibt, kann man die Sprache nicht lernen. Forth ist eine Form des gesteigerten ClosedSource, wo also nicht der Quellcode a) geheim ist und b) gar nicht als Quellcode niedergeschrieben wird. Das man damit Informatik betreiben kann, demonstriert die Forth Community eindrucksvoll. Auf den Treffen werden immer mal wieder Algorithmen vorgestellt die etwas sinnvolles tun. Wie man zu diesen Algorthmen kommt ist unklar, vermutlich durch mündliche Überlieferung. Vielleicht gibt es auch irgendwo Aufzeichnungen darüber die jedoch nicht so offensichtlich sind. Die Grundhaltung ist jedoch restriktiv ausgelegt. Forth bedeutet Informatik zu betreiben ohne darüber zu publizieren und ohne den Sourcecode aufzuschreiben. Das heißt, man betrachtet das Wissen über Computer als zu wertvoll um es mit der Welt zu teilen sondern schreibt es wie Leonardo da Vinci in Spiegelschrift auf oder noch besser, hinterlässt keinerlei Notizen. Damit vermeidet man, dass das Wissen in Falsche Hände gelangt. So ähnlich wie in dem Film “Der Name der Rose” als der halbblinde Mönch ein wichtiges Buch versteckt hat.

Und hier wird auch deutlich warum die Forth Community so abfällig über UNIX und die C-Programmiersprache urteilt. Weil UNIX darauf ausgelegt ist, Informationen zu teilen. Also nach der Hacker-Mentalität Sourcecode auf Tape überall hinzusenden, und in Standardbüchern alles einsteigerfreundlich aufzubereiten so dass es jeder versteht. UNIX und C ist von der Mentalität her das genaue Gegenteil von Forth. Die einen Teilen Wissen, die anderen behalten es für sich. Die einen schreiben es in Büchern und als Quelltext nieder, die anderen löschen den Code wieder nachdem der Computer den Task ausgeführt hat. Die einen wollen mit dem Computer die Welt verändern die anderen wollen zurück ins Mittelalter.

Forth ist keine Programmiersprache oder Programmiersystem, Forth ist der Gegenentwurf zur Schriftkultur. Es geht nicht darum, Anfänger an das Programmieren heranzuführen oder Betriebssysteme mit anderen zu teilen sondern es geht um Macht. Na ja, vielleicht kann man es auch etwas vorsichtiger so formulieren, dass es nur um die Kosten geht. Wenn man den Sourcecode eines Programms nicht in einer Datei speichert sondern nur im Kopf herumträgt dann steigen die Kosten an. Will jemand anderes das Programm ausführen kann er die Software nicht einfach bei github herunterladen. Er kommt trotzdem an den Code, muss aber die Gegenseite vorher darum bitten. Man kommt trotzdem zum Ziel, nur ist es aufwendiger.

Die Tendenz Informationen für sich zu behalten ist auch außerhalb von Forth ein weit verbreitetes Ärgerniss. Der Sourcecode zu Windows 10 steht bis heute nicht auf github und wie ROS im Detail funktioniert, darüber gibt es leider keine Handbücher wo man es nachlesen kann. Gewissermaßen ist das Fehlen von Sourcecode und das

Fehlen von Handbüchern der Normalfall was man aktiv überwinden muss um die Dinge voranzubringen. Geschichtlich gesehen stellt die Erfindung von OpenSource eine Zäsur da. Nicht weil plötzlich bessere Software bereitstand sondern weil Software als Medium verstanden wurde, was ähnlich wie das Fernsehen in jeden Haushalt übertragen wird.

Von diesem Ideal ist die Forth Community meilen weit entfernt. Es gibt noch nichtmal Libraries in Forth die man teilen könnte. Auch Bücher wo Forth Programmierung erklärt wird sind Mangelware. Ja richtig, Forth ist Mangelwirtschaft, es fehlt an allem und das wenige was es gibt ist sehr teuer.

Astrologie Es gibt einen Kontext wo es sinnvoll ist Dinge geheimzuhalten. Und zwar wenn man sich selbst als Zauberkünstler versteht wo es zum Berufsethos gehört, keinen Trick zu verraten. Ein typisches Forth Projekt könnte in einem Astrologieprogramm bestehen, oder vielleicht auch eine Software zum Gedankenlesen. Ob diese Software wiklich etwas sinnvolles tut, oder man nicht einfach nur die Gegenseite betrügt sei mal dahingestellt, aber mit Forth geht sowas ausgezeichnet. So nach dem Motto: mein Forth Programm sagt mir, dass du jetzt müde wirst, dich hinlegst, deine Gedanken werden ruhig. Im Bereich des Varietes sind solche Zauberdarbietungen sehr beliebt. Und die Zuschauer erwarten genau diese Form der Darbietung. Sie wollen einen richtigen Magier sehen der mit ausholenden Gesten die Geister beschwört und ihnen den Untergang verkündet. Auf der Weltausstellung 1939 gab es den Roboter Electro zu bestaunen. Damals war Forth zwar noch nicht erfunden, aber es wäre die ideale Sprache gewesen um den Roboter zu steuern. Und zwar weil Elektro wie auch Forth auf maximale Ehrfurcht hin erstellt wurden. Es ging bei Elektro darum von oben herab zu predigen. Also modernen Schamanismus zu betreiben.

Elektro damals und das heutige Forth darf man nicht als technische Errungenschaft interpretieren sondern es handelt sich um eine Theateraufführung. Das heißt, Elektro funktioniert nur, wenn es Zuschauer gibt, die von seiner gigantischen Stimme eingeschüchtert werden und Forth funktioniert nur, wenn jemand es programmieren möchte. Es geht nicht wirklich um Programmieren sondern um das Setting wie Programmieren unterrichtet wird. Am besten kann man Forth in einer Schule um 1900 unterrichten wo es noch die Prügelstrafe gab, nicht selbstgesteuertes Lernen war damals angesagt sondern Gehorsam dem Lehrkörper gegenüber. Man darf nicht den Fehler machen, Forth in Frage stellen zu wollen, weil das eine unangenehme soziale Situation erzeugt bei dem der Lehrer seine Autorität durchsetzen wird. Mit Forth kommunzieren meint, das soziale Setting zu erkennen in dem die Sprache gesprochen wird.

11.11 Ist Forth tot?

Zwischen Forth und Lisp gibt es große Gemeinsamkeiten. Lisp ist eine Sprache der künstlichen Intelligenz, genauer gesagt war es eine. Wer sich heute mit LISP beschäftigen möchte, recherchiert im Idealfall in Computermuseen. Er findet dort Lisp in Hardware, die alten Handbücher zu Common Lisp und wird sogar entdecken dass auch die DDR Mitte der 1980'er Jahre angefangen hat sich für LISP zu interessieren. Rückblickend war das natürlich reine Utopie. Die Computer der damaligen Zeit waren nicht leistungsfähig genug für das was man vorhatte, es gab keine Software und die Leute vor den Terminals waren schlecht informiert und lausige Programmierer. Das ist kein Werturteil, sondern ein Rückblick auf die 1980'er Jahre.

Und was für LISP gilt das gilt auch für Forth. Forth ist lediglich stärker auf die Maschinenebene hin zugeschnitten. Will man darin größere Programme schreiben funktioniert es ähnlich wie bei LISP auch. Aber warum genau hat sich beides nicht durchsetzen können, warum stehen die LISP Maschinen heute in einem Museum? Zunächst einmal nützt die Programmiersprache gar nichts, wenn der Arbeitsspeicher zu klein ist. Installiert man auf einem Commodore 64 einen LISP Interpreter hat man in der Theorie einen leistungsfähigen Heimcomputer der 5. Computergeneration der optmal geeignet ist zu Robotiksteuerung und für komplexes Theorembeweisen, doch faktisch hat man nicht mehr als 40 kb freien Arbeitsspeicher und wenn man dort zwei Seiten eintippt gibt es einen Stackoverflow.

Das zweite Problem mit LISP ist, dass es zwar eine elgante Programmiersprache ist, die selbstverständlich Metaprogrammierung, das Parsen von weiteren Sprachen und sogar Objektorientierung unterstützt nur reicht das leider nicht. Benötigt man ernsthafte Anwendungen kommt es nicht auf Sprachfeatures an sondern auf die Menge an Codezeilen. Der Rechner nützt gar nichts, wenn man begleitend kein Software-Entwicklungsteam hat was ihn programmiert. Die Mächtigkeit von Computern definiert sich nicht aus der Maschine selbst sondern was die Gesellschaft darumherum erfindet. Microsoft ist ein Beispiel dafür, oder LibraOffice. Beides sind keine Programmiersprachen sondern es sind Organisationen die Software entwickeln. Man kann es wie folgt zuzusammenfassen. Die Programmiersprche BASIC gilt als hoffnungslos. Die Sprache unterstützt weder Prozeduren und läuft extrem langsam. Aber, würde man um BASIC herum eine Softwareindustrie aufbauen wo 10000 Programmierer Code entwickeln könnte man in BASIC leistungsfähige CAD Anwendungen, Robotik-Steuerungen und Bilderkennung realisieren. Das wurde zwar nie gemacht, stattdessen wurde das Ökosystem um C/C++ herum errichtet weil diese Sprachen als effizienter gelten, aber worum es geht ist, dass Computer und Programmiersprachen nur ein Medium sind was erstnoch befüllt werden muss.

Und nicht viel anders ist es mit Forth. Forth ist eine Art von Metaprogrammiersprache. Also ein System mit dem man andere Programmiersprache und andere Betriebssysteme entwickeln kann. Ein sehr leistungsfähiges Metasystem sogar. Nur leider reicht das nicht aus. Die Grenze von Forth wird weniger durch die MIPS Zahl der CPU bestimmt, sondern durch den verfügbaren Sourcecode. Wenn man keine Firmen findet, die Software für Forth programmieren nützt die Sprache gar nichts. Forth selber ist nicht tod, aber die Community darum herum. Damit ist gemeint, dass es keine Firmen gibt, die in Forth Betriebssysteme erstellen oder Datenbanken programmieren.

Früher war das einmal anders. In den 1980'er war zusammen mit LISP die Blütezeit von Forth. Damit ist gemeint, dass in dieser Zeit noch nicht klar war, welche Sprachen sich einmal durchsetzen werden. n den 1980'er wurden Forth und C gleichberechtigt diskutiert. Der Grund war, dass es damals generell keine Softwarefirmen gab. Software wurde damals nur an Universitäten programmiert und dort in LISP, Forth und ähnlichen Sprachen.

Im Jahr 1976 hat Bild Gates einen Brief geschrieben. Er hat darin keine neue Programmiersprache angekündigt und auch kein Betriebssystem. Sondern Bill Gates forderte in dem Brief die Hackercommunity auf, für Software Geld zu bezahlen. Diese Ideologie war die eigentliche Technologie welche die Informatik vorangebracht hat. Es geht nicht so sehr darum, den Hauptspeicher zu organisieren oder LISP Code zu schreiben sondern worum es geht ist Business. Also Leute anzustellen die Code schreiben und Firmen zu gründen die daraus ein Produkt machen. Machmal wird Bill Gates als Gegenspieler zur OpenSource Bewegung gesehen. Aber das ist nicht wahr. Eigentlich ist das Geschäftsmodell von Red hat und das von Microsoft sehr ähnlich. Der Unterschied ist nur, dass bei Red Hat alles verkauft wird inkl. dem Sourcecode. An der Idee, Software als Business zu betreiben hat sich nichts geändern.

Manchmal wird gesagt, man könnte auf einer LISP Maschine Software entwickeln. Nach dem Einschalten gibt es eine GUI mitsamt

Emacs Texteditor zu sehen. Genau genommen ist die LISP Maschine jedoch wertlos. Sie ist vergleichbar mit einer leeren Filmrolle. Um Software zu entwickeln braucht man Programmierer. Eigentlich braucht man nichts anderes als Programmierer, auf eine LISP Maschine kann man zur Not auch verzichten. Bei Microsoft arbeitet man beispielsweise nicht mit LISP Maschinen sondern mit etwas anderem und entwickelt auch Software. Was man eigentlich benötigt ist eine Community. Also Leute die sich zusammenschließen und Code schreiben. Man kann das auf Basis von LISP tun aber auch mit C++, C# oder sonstwie.

Die entscheidene Einsicht lautet, dass Metaprogramming nicht funktioniert. Metaprogramming ist ein Konzept aus den 1970'er wonach Software sich von alleine programmiert. Das man also ein Programm schreibt was aus 1000 Zeilen Code besteht und das erzeugt dann ein Programm was 10000 Zeilen lang ist. Warum Metaprogramming damals als Vision gehandelt wurde hat etwas mit der Computergeschichte zu tun. Genau gesagt, waren die ersten Compiler und Hochsprachen Metaprogrammier-Tools. Man hat folglich geglaubt, dass man damit insgesamt Software entwickeln könnte. Das war ein Irrtum. Als höchste Form der Programmiersprachen gelten Sprachen wie C++, also objektorientierte Sprache. Eine Abstraktionsstufe höher geht es nicht mehr. Es gibt nichts womit man das Programmieren drastisch erleichtern kann. Und man kann sogar C++ durch ein simples C ersetzen ohne dass die Produktität stark leidet. Nur, ein Compiler alleine ist noch kein Programm, nur wenn man Programmierer findet die damit Code schreiben kann man die Systeme hochskalieren. Also mit dem Computer Spiele spielen, Zahlen verwalten oder Roboter steuern.

Und genau hier kommt die Vision von JCR Licklider ins Spiel. Er hat relativ früh erkannt, dass Künstliche Intelligenz grenzen hat. Man kann zwar in Software definieren wie man ein Fortran Programm in Assemblycode umwandelt aber man kann nicht in Software vorgeben wie man Bilder malt oder eigene Programme schreibt. Was man tun kann ist den Computer als Medium zur Vernetzung zu verwenden. Also über das Internet die Programmierer zusammenbringen damit die dann Code schreiben. Das ist keine richtige Künstliche Intelligenz sondern es ist computerunterstütztes Programmieren.

Literatur

[1] Darren M Chitty. Faster gpu-based genetic programming using a two-dimensional stack. *Soft Computing*, 21(14):3859–3878, 2017.

[2] M Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen a generator of efficient virtual machine interpreters. *Software: Practice and Experience*, 32(3):265–294, 2002.

[3] Ruijie Fang and Siqi Liu. A performance survey on stack-based and register-based virtual machines. *arXiv preprint arXiv:1611.00467*, 2016.

[4] The flux Research Group. The oskit the flux operating system toolkit 0.97. *University of Utah https://www.cs.utah.edu/flux/oskit/doc/oskit.ps.gz*, 2002.

[5] Stefan Forstenlechner, Miguel Nicolau, David Fagan, and Michael ONeill. Grammar design for derivation tree based genetic programming systems. In *European Conference on Genetic Programming*, pages 199–214. Springer, 2016.

[6] Paul Frenger. Is forth dead? *ACM Sigplan Notices*, 36(6):23–25, 2001.

[7] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1828–1836, 2015.

[8] Philip Koopman. Stack computers: the new wave. 1989.

[9] Charles Eric LaForest. Second-generation stack computer architecture. B.S. thesis, University of Waterloo, 2007.